STRATÉGIE DE LOGS EN C#/.NET

STOP AUX `Error` POUR DU FONCTIONNEL

Ressources & version officielle : nicolas-cousin.com

SYMPTÔME

- Des règles métier génèrent des `Error`
- Conséquence : alertes polluées, équipes désensibilisées, enquêtes plus lentes

RÈGLE D'OR (NIVEAU = ACTION ATTENDUE)

- `Information` : événement normal
- `Warning` : anomalie attendue / récupérable
- `Error` : échec non récupérable ou impact SLO
- `Critical` : indisponibilité, sécurité, corruption

FONCTIONNEL VS TECHNIQUE

Fonctionnel → souvent `Warning/Information`

- Règle métier non satisfaite, doublon, idempotence

Technique → `Error/Critical`

- Exception non gérée, DB down, timeout, corruption

EXEMPLES – FONCTIONNEL

```
logger.LogWarning("Payment refused: insufficient funds. OrderId={
    orderId, customerId);

logger.LogInformation("Order already processed (idempotent). Orde
```

EXEMPLES – TECHNIQUE

```
catch (SqlException ex)
{
    logger.LogError(ex, "Database failure while confirming order.
    throw;
}
```

LOGS NON STRUCTURÉS VS STRUCTURÉS

Problème avec les logs non structurés

- Concaténation de strings → difficile à parser/filtrer
- Impossible de requêter efficacement sur des valeurs spécifiques
- Perte de typage et de contexte

❌ EXEMPLE – LOG NON STRUCTURÉ

```
// ❌ Mauvais : concaténation de strings
logger.LogWarning("Payment refused for order " + orderId +
    " and customer " + customerId + " due to insufficient funds")

logger.LogError("Database connection failed while processing orde
    orderId + " at " + DateTime.Now);
```

✅ EXEMPLE – LOG STRUCTURÉ

```
// ✅ Bon : propriétés structurées
logger.LogWarning("Payment refused: insufficient funds. OrderId={
    orderId, customerId);

logger.LogError(ex, "Database connection failed. OrderId={OrderId
    orderId);
```

VUE DANS UN SYSTÈME DE LOGS CENTRALISÉ

❌ Log non structuré (Datadog/Elasticsearch/Seq)

```
{
  "timestamp": "2026-02-04T10:15:23.456Z",
  "level": "Warning",
  "message": "Payment refused for order 12345 and customer C-789
}
```

Problème : Impossible de filtrer par `OrderId=12345` ou `CustomerId=C-789`

VUE DANS UN SYSTÈME DE LOGS CENTRALISÉ

✅ Log structuré (Datadog/Elasticsearch/Seq)

```
{
  "timestamp": "2026-02-04T10:15:23.456Z",
  "level": "Warning",
  "message": "Payment refused: insufficient funds",
  "OrderId": 12345,
  "CustomerId": "C-789"
}
```

Filtres possibles : `OrderId:12345`, `CustomerId:"C-789"`, dashboards par client

LOGS STRUCTURÉS + `EventId`

- Propriétés obligatoires : OrderId, CustomerId, CorrelationId/TraceId
- `EventId` pour classer, filtrer, dashboarder
- Pas de concaténation de strings

EXEMPLE EventId

```
private static readonly EventId PaymentRefused = new EventId(1201
private static readonly EventId DbFailure      = new EventId(5001

logger.LogWarning(PaymentRefused,
    "Payment refused. Reason={Reason} OrderId={OrderId}", reason,

logger.LogError(DbFailure, ex,
    "Database failure. OrderId={OrderId}", orderId);
```

EXCEPTIONS & NIVEAUX DE LOG

Exception ≠ toujours une "erreur"

- C'est un mécanisme technique de signalement
- Le niveau de log reflète l'impact et l'action attendue

NIVEAUX DE LOG : ACTION ATTENDUE

`Warning` = situation métier à surveiller / analyser

- Règle métier non satisfaite, refus, validation
- Action : suivi, analyse, amélioration fonctionnelle

`Error` = incident technique / dysfonctionnement

- DB indisponible, bug, timeout non récupérable
- Action : investigation technique immédiate, escalade si nécessaire

POURQUOI ÉVITER ERROR POUR LE FONCTIONNEL

Trop d'Error = perte de confiance

- Alertes noyées dans le bruit fonctionnel
- Équipes désensibilisées → réactions plus lentes
- Difficulté à distinguer vrais incidents des cas métier

Conséquence opérationnelle

- Les vraies pannes techniques passent inaperçues
- Temps de détection et de résolution allongé
- Impact sur SLO et disponibilité

COMPROMIS MOA/MOE (GAGNANT-GAGNANT)

Objectif MOA : visibilité + pilotage

- Suivre les irritants fonctionnels (refus, rejets, validations)
- Piloter l'amélioration continue du métier

Objectif MOE/OPS : signal fiable

- Détecter rapidement les incidents techniques
- Minimiser les fausses alertes et le bruit

PROPOSITION DE COMPROMIS

Fonctionnel attendu/récupérable

- `Warning` ou `Information` + `EventId` + propriétés structurées
- Dashboards dédiés pour le suivi métier
- Alertes possibles sur Warning ciblés (volume, seuil, tendance)

Technique/non récupérable

- `Error` ou `Critical` + alerte immédiate
- Escalade automatique vers l'équipe technique

EXEMPLE CONCRET

Visibilité métier sans Error

- Dashboard "Taux de refus de paiement" (Warning)
- Dashboard "Validations échouées par règle" (EventId)
- Alerte si taux > seuil pendant X minutes

Alertes techniques fiables

BEST PRACTICES : EXCEPTIONS & LOGGING

Éviter les exceptions pour le flux métier normal

- Préférer validation explicite, pattern Result/Try
- Exception = cas exceptionnel, pas règle métier

BEST PRACTICES (SUITE)

Catch ciblé

- Attraper des exceptions spécifiques (`SqlException`, `HttpRequestException`)
- Éviter `catch(Exception)` hors "frontière" (middleware global)

Logging unique

- Éviter le double log (local + global)
- Décider où l'exception est loggée (une seule fois)

BEST PRACTICES (SUITE)

Rethrow correct

- Utiliser `throw;` (pas `throw ex;`) pour garder la stacktrace

BEST PRACTICES (SUITE)

Logs structurés

- Toujours inclure les identifiants métier (OrderId, CustomerId)
- Inclure CorrelationId/TraceId pour traçabilité

EventId standardisé

- Permet le suivi (KPI, requêtes, dashboards)
- Convention : 12xxx = business, 50xxx = technique

BEST PRACTICES (SUITE)

Données sensibles

- Éviter PII dans les logs
- Masquer ou anonymiser les données personnelles

EXEMPLES DE CODE

Illustration du passage "MOA-intent" vers "Best practice"

❌ AVANT : EXCEPTION MÉTIER → ERROR

```
public async Task<OrderResult> ProcessOrder(int orderId)
{
    try
    {
        var order = await _orderRepository.GetById(orderId);
        if (order.Amount > customer.Balance)
        {
            throw new InsufficientFundsException("Customer has in
        }
        // ...
    }
    catch (InsufficientFundsException ex)
    {
        _logger.LogError(ex, "Order processing failed. OrderId={O
        return OrderResult.Failed("Insufficient funds");
    }
```

✅ APRÈS : VALIDATION EXPLICITE → WARNING

```
private static readonly EventId InsufficientFunds = new EventId(1

public async Task<OrderResult> ProcessOrder(int orderId)
{
    var order = await _orderRepository.GetById(orderId);
    var customer = await _customerRepository.GetById(order.Custom

    if (order.Amount > customer.Balance)
    {
        _logger.LogWarning(InsufficientFunds,
            "Payment refused: insufficient funds. OrderId={OrderI
            orderId, customer.Id, order.Amount, customer.Balance)
        return OrderResult.Failed("Insufficient funds");
    }

    // Process order
```

MAPPING SIMPLE : BUSINESS → WARNING

```
// Cas métier attendus → Warning/Information
if (!IsValidInput(input))
{
    _logger.LogWarning(ValidationFailed,
        "Input validation failed. OrderId={OrderId} Reason={Reaso
        orderId, validationResult.Reason);
    return Result.Invalid();
}

if (await _orderRepository.Exists(orderId))
{
    _logger.LogInformation(OrderAlreadyProcessed,
        "Order already processed (idempotent). OrderId={OrderId}"
        orderId);
    return Result.AlreadyProcessed();
}
```

MAPPING SIMPLE : TECHNIQUE → ERROR

```
// Cas technique/infrastructure → Error
catch (SqlException ex)
{
    _logger.LogError(DatabaseFailure, ex,
        "Database connection failed. OrderId={OrderId}",
        orderId);
    throw; // Rethrow pour préserver stacktrace
}

catch (HttpRequestException ex) when (ex.StatusCode == HttpStatus
{
    _logger.LogError(ExternalApiTimeout, ex,
        "External API timeout. OrderId={OrderId} Endpoint={Endpoi
        orderId, apiEndpoint);
    throw;
}
```

MIDDLEWARE GLOBAL : BOUNDARY ASP.NET CORE

```
public class GlobalExceptionMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<GlobalExceptionMiddleware> _logger;

    public GlobalExceptionMiddleware(RequestDelegate next, ILogge
    {
        _next = next;
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
```

VARIANTES SELON LE RUNTIME

Présentation rapide de la même logique avec le bon style pour chaque couple .NET/C#.

.NET FRAMEWORK 4.8 – C# 7.3

```
private static readonly EventId PaymentRefused = new EventId(1201
private static readonly EventId DbFailure      = new EventId(5001

logger.LogWarning(PaymentRefused,
    "Payment refused. Reason={Reason} OrderId={OrderId}", reason,

logger.LogError(DbFailure, ex,
    "Database failure. OrderId={OrderId}", orderId);
```

.NET 8 – C# 12

```
private static readonly EventId PaymentRefused = new(12010, nameo
private static readonly EventId DbFailure      = new(50010, nameo

logger.LogWarning(PaymentRefused,
    "Payment refused. Reason={Reason} OrderId={OrderId}", reason,

logger.LogError(DbFailure, ex,
    "Database failure. OrderId={OrderId}", orderId);
```

EXCEPTIONS : SIMPLE RÈGLE

- Cas métier gérés **sans** exception
- Exceptions réservées à l'imprévu technique

MINI-CHECKLIST (ACTIONNABLE EN ÉQUIPE)

1. Table “niveau par scénario” (fonctionnel vs technique)
2. Catalogue `EventId` + propriétés standardisées
3. Alerter surtout sur `Error/Critical` (Warnings ciblés au besoin)
4. Revue Top 20 `Error` → downgrade si fonctionnel

MESSAGE FINAL

- `Warning` = incident **fonctionnel** maîtrisé / récupérable
- `Error` = incident **technique** ou échec non récupérable
- Moins de bruit → meilleures alertes → moins de temps perdu