

# Análisis de Performance - API Tokenización

Durante el análisis se identificaron **problemas críticos de rendimiento** que comprometen la capacidad de la API para procesar múltiples peticiones concurrentes. Estos hallazgos afectan directamente la escalabilidad del servicio y pueden impactar a los clientes consumidores especialmente bajo condiciones de alta demanda.

## Resumen de Problemas Identificados

Severidad	Problema	Descripción	Impacto
<div><div></div>Crítico</div>	Bloqueos con .Result/.Wait/ .GetAwaiter().GetResult()	Uso extensivo de llamadas bloqueantes en código asíncrono	Deadlocks, degradación severa de performance, agotamiento del ThreadPool
<div><div></div>Crítico</div>	HttpClient Sin Polly	Llamadas HTTP sin políticas de retry ni circuit breaker	Fallos en cascada, timeouts no manejados, experiencia degradada
<div><div></div>Crítico</div>	MongoClient con ciclo Transient	MongoClient se crea nuevo en cada request en lugar de ser Singleton	Handshake SSL/TLS repetido, agotamiento de conexiones, overhead extremo bajo carga
<div><div></div>Crítico</div>	Estado Global Sin Thread-Safety	Diccionarios estáticos modificables sin sincronización	Data races, condiciones de carrera, corrupción de datos, comportamiento no determinista
<div><div></div>Alta</div>	Newtonsoft.Json Heredado	Uso de Newtonsoft.Json en lugar de System.Text.Json	Serialización 2-3x más lenta, mayor asignación de memoria, GC más frecuente
<div><div></div>Alta</div>	SemaphoreSlim Innecesario	Límite artificial de 300 operaciones simultáneas en MongoDB	Cuello de botella innecesario, latencia adicional en cada query, reduce throughput
<div><div></div>Alta</div>	Servicios DI Sin Keyed Services	Múltiples implementaciones registradas sin usar keyed services (.NET 8)	Resuelve todas las implementaciones, desperdicio de CPU/memoria, código menos mantenible

# Hallazgos Detallados

## 1. Async/Await & ThreadPool

### 1.1 Sync-over-Async: `.Result` en `RestClientService`

Severidad: ● Crítica

Ubicación:

- `Dev_Resources/SharedKernel/Services/RestClientService.cs` líneas 138, 148, 158, 165, 179

Impacto:

- Bloquea hilos del ThreadPool esperando I/O de red (renovación de tokens).
- Bajo carga concurrente, agota worker threads causando timeouts y degradación de latencia.
- El ThreadPool puede saturarse completamente.

Evidencia:

El archivo `RestClientService.cs` contiene 5 métodos HTTP ( `Delete` , `Get` , `Post` x2, `Put` ) que utilizan el patrón problemático `.Result` , el cual bloquea el thread actual mientras espera la respuesta de `PrepareCallAsync` .

El método `PrepareCallAsync` es responsable de:

1. Realizar llamadas HTTP externas para obtener tokens OAuth
2. Autenticar las peticiones antes de ejecutar la operación solicitada

**Problema:** Al usar `.Result` en lugar de `await` , se convierte una operación asíncrona (no bloqueante) en una operación síncrona (bloqueante), desperdiciando recursos del ThreadPool y reduciendo la capacidad de la API para manejar múltiples peticiones simultáneas.

Remediación:

⚠ Problema:

```
public Task<HttpResponseMessage> Delete(string service, string path)
{
    var restClientServiceSetting = PrepareCallAsync(service, path).Result; ⚠
    string endpoint = $"{restClientServiceSetting.BaseAddress}{path}";
    return restClientServiceSetting.Client!.DeleteAsync(endpoint);
}
```

✅ Solución:

```
public async Task<HttpResponseMessage> Delete(string service, string path)
{
    var restClientServiceSetting = await PrepareCallAsync(service, path);
    string endpoint = $"{restClientServiceSetting.BaseAddress}{path}";
    return await restClientServiceSetting.Client!.DeleteAsync(endpoint);
}
```

## 1.2 Sync-over-Async: `.Result` en `CustomerTokenValidator`

Severidad:  Crítica

Ubicación:


- `Dev_Resources/Application/Validator/CustomerTokenValidator.cs` línea 22

Impacto:

- El método `.Result` bloquea el thread actual esperando que termine la operación async.
- Se ejecuta en cada validación de `CustomerToken` en el controller.
- El thread bloqueado no puede procesar otros requests mientras espera.
- Con alta concurrencia, se agotan los threads del `ThreadPool`.

Remediación:


 Problema:

```
public static CustomerTokenGetterResponse ContinueOrThrowException(
    IEnumerable<ITokenizationGetterService> tokenizationGetterService,
    string customerToken)
{
    var customerGetterTokenization = tokenizationGetterService.FirstOrDefault(x => x is CustomerGetterService);
    var result = customerGetterTokenization?
        .GetTokenData<CustomerTokensEntity, CustomerTokenGetterResponse>(customerToken).Result 
        ?? throw new NotFoundException();
    return result;
}
```

 Solución:

```
public static async Task<CustomerTokenGetterResponse> ContinueOrThrowExceptionAsync(
    IEnumerable<ITokenizationGetterService> tokenizationGetterService,
    string customerToken)
{
    var customerGetterTokenization = tokenizationGetterService.FirstOrDefault(x => x is CustomerGetterService);
    var result = await customerGetterTokenization!
        .GetTokenData<CustomerTokensEntity, CustomerTokenGetterResponse>(customerToken)
        ?? throw new NotFoundException();
    return result;
}
```

### 1.3 Sync-over-Async: `.Result` en `CardCreatorService`

Severidad:  **Alta**

Ubicación:

- `Dev_Resources/Application/Services/CardCreatorService.cs` línea 66

Impacto:

- Igual que el hallazgo 1.2: bloquea el thread esperando una operación async.
- Se ejecuta en cada creación de `CardToken`.
- Reduce el throughput de la API bajo alta carga.

Remediación:


 **Problema:**

```
private protected override object MapResponseObject(object request, string dataToken)
{
    CardTokenDto cardTokenDto = (CardTokenDto)request;
    CardTokenResponse result = _mapper.Map<CardTokenResponse>(cardTokenDto);
    result.CardToken = dataToken;
    TiposProducto card = GetProductId(cardTokenDto.CardNumber, Convert.ToInt32(cardTokenDto.TenantId, cultures)).Result; ⚠
    result.ProductID = card.Nombre!;
    result.Franchise = card.Franquicia!;
    return result;
}
```

 **Solución:**

```
private protected override async Task<object> MapResponseObjectAsync(object request, string dataToken)
{
    CardTokenDto cardTokenDto = (CardTokenDto)request;
    CardTokenResponse result = _mapper.Map<CardTokenResponse>(cardTokenDto);
    result.CardToken = dataToken;
    TiposProducto card = await GetProductId(cardTokenDto.CardNumber,
        Convert.ToInt32(cardTokenDto.TenantId, CultureInfo.CurrentCulture));
    result.ProductID = card.Nombre!;
    result.Franchise = card.Franquicia!;
    return result;
}
```

## 1.4 Sync-over-Async: `Task.Run().Wait()` en Middleware

Severidad:  Alta

Ubicación:


- `Dev_Resources/WebApi/Extensions/ExceptionHandlerMiddleware.cs` línea 113


Impacto:

- Ejecutado en manejo de excepciones; agrega latencia adicional a respuestas de error.
- `Task.Run` introduce overhead innecesario (cola en ThreadPool).

Remediación:

 Problema:

```
Task.Run(() => repository.InsertAsync(ex)).Wait(); 
```

 Solución:

```
private static async Task SaveError(ILogger<ExceptionHandlerMiddleware> logger, IRepository repository, ExceptionLog ex)
{
    try
    {
        await repository.InsertAsync(ex);
    }
    catch (Exception exMongo)
    {
        logger.LogError(exMongo, "Error persisting exception to database");
    }
}
```

**Nota Importante:** El método `SaveError` debe ser eliminado completamente. Después de conversaciones con el equipo de desarrollo y validación con los canales consumidores, se determinó que persistir excepciones en la base de datos no aporta valor dado que la aplicación ya cuenta con **Instana** como herramienta de monitoreo y observabilidad. Instana proporciona trazabilidad completa de errores, métricas en tiempo real y correlación de eventos, haciendo redundante e ineficiente el almacenamiento de logs de excepciones en MongoDB. Esta eliminación además reducirá la carga en la base de datos y mejorará el rendimiento del middleware de manejo de errores.

## 2. HttpClient & Polly

### 2.1 Ausencia de políticas de resiliencia

Severidad:  Crítica

Ubicación:

- `Dev_Resources/WebApi/Extensions/ServiceExtensions.cs:41` – solo `AddHttpClient()` sin configuración.


Impacto:

- Si un servicio externo tiene un error temporal (timeout, 503), la API retorna error 500 inmediatamente sin reintentar.
- Si un servicio externo está caído, cada request intenta conectarse sin límite de tiempo, bloqueando threads.
- Si un servicio externo responde lento, no hay timeout configurado, las conexiones HTTP se pueden agotar.
- Sin circuit breaker: si un servicio está caído, seguimos enviando requests que van a fallar.

Evidencia:

- No hay referencias a Polly en el código.
- `RestClientService` no configura `Timeout` ni handlers de política.

Remediación:

 Problema:

```
// ServiceExtensions.cs
services.AddHttpClient(); // Sin configuración
```


 Solución:

```
// Instalar primero: dotnet add package Microsoft.Extensions.Http.Polly

services.AddHttpClient("TokenClient")
    .AddTransientHttpErrorPolicy(p => p.WaitAndRetryAsync(3, retryAttempt =>
        TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))))
    .AddTransientHttpErrorPolicy(p => p.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)))
    .ConfigureHttpClient(client => client.Timeout = TimeSpan.FromSeconds(10));
```

**Nota Importante:** La parametrización debe ser definida según las necesidades de cada cliente y su carga de trabajo esperada. Es crucial realizar pruebas de carga y ajustar los valores de timeout, retry y circuit breaker en base a los resultados obtenidos.

## 2.2 HttpClient creado manualmente en RestClientService

Severidad:  Alta

### Ubicación:

- `Dev_Resources/SharedKernel/Services/RestClientService.cs` línea 40

### Impacto:

- El HttpClient creado no tiene configuración de timeout, retry ni circuit breaker.
- No se pueden aplicar políticas de resiliencia centralizadas.
- El pooling de conexiones HTTP no se optimiza correctamente.

### Remediación:

#### Problema:

```
// RestClientService.cs
restClientServiceSetting.Client = _clientFactory.CreateClient(); // Sin configuración
```

#### Solución:

```
// ServiceExtensions.cs
services.AddHttpClient<IRestClientService, RestClientService>("RestClient")
    .ConfigurePrimaryHttpMessageHandler(() => new SocketsHttpHandler
    {
        PooledConnectionLifetime = TimeSpan.FromMinutes(5),
        PooledConnectionIdleTimeout = TimeSpan.FromMinutes(2),
        MaxConnectionsPerServer = 100
    });
```

## 3. MongoDB & EF Core

### 3.1 MongoClient con ciclo de vida Transient

Severidad:  Crítica


Ubicación:

- `Dev_Resources/WebApi/Extensions/MongoRepositoryExtensions.cs` línea 20

Impacto:

- MongoClient se crea nuevo en cada request (debería ser una sola instancia compartida para toda la aplicación).
- Cada conexión nueva hace handshake SSL/TLS y autenticación contra MongoDB (proceso costoso).
- Con 500 requests por segundo, se crean 500 MongoClients nuevos por segundo.
- MongoDB tiene un límite de conexiones simultáneas, este patrón puede agotar el pool de conexiones.

Remediación:

 Problema:

```
// MongoRepositoryExtensions.cs
services.AddTransient<IMongoDatabase>(sp =>
{
    MongoClient client = new(configuration.MongoDbConnectionString); // Nueva instancia cada vez
    return client.GetDatabase(configuration.MongoDataBaseName);
});
```


 Solución:

```
// MongoRepositoryExtensions.cs
services.AddSingleton<IMongoClient>(sp =>
    new MongoClient(configuration.MongoDbConnectionString));

services.AddScoped<IMongoDatabase>(sp =>
{
    var client = sp.GetRequiredService<IMongoClient>();
    return client.GetDatabase(configuration.MongoDataBaseName);
});
```



## 3.2 SemaphoreSlim innecesario limitando concurrencia

Severidad:  Media

### Ubicación:

- `Dev_Resources/Persistence/Mongo/Repository.cs` líneas 23, 38, 77

### Impacto:

- El SemaphoreSlim limita a 300 operaciones simultáneas de MongoDB, pero este límite es arbitrario.
- El `Driver` de MongoDB ya tiene su propio pool de conexiones interno que maneja la concurrencia correctamente.
- Cada operación debe hacer `await _semaphoreSlim.WaitAsync()` antes de acceder a MongoDB (overhead innecesario).
- Si llegan 301 requests simultáneos, el request #301 se bloquea esperando que se libere el semáforo.
- De conservarse el `SemaphoreSlim`, el Repository no implementa `IDisposable` para liberar el semáforo al destruirse.
- Cada operación agrega latencia del `WaitAsync()` sin beneficio real.

### Remediación:

#### ⚠ Problema:

```
// Repository.cs
public class Repository : IRepository
{
    private readonly SemaphoreSlim _semaphoreSlim = new(300,300);

    public async Task<TEntity> InsertAsync<TEntity>(TEntity entity) where TEntity : BaseEntity
    {
        await _semaphoreSlim.WaitAsync(); // Limita a 300 operaciones
        try
        {
            IMongoCollection<TEntity> collection = _mongoDatabase.GetCollection<TEntity>(typeof(TEntity).Name);
            await collection.InsertOneAsync(entity);
            return entity;
        }
        finally
        {
            _semaphoreSlim.Release();
        }
    }
}
```


#### ✅ Solución:

```
// Repository.cs - Eliminar SemaphoreSlim completamente
public class Repository : IRepository
{
    private readonly IMongoDatabase _mongoDatabase;
    // ✗ ELIMINAR: private readonly SemaphoreSlim _semaphoreSlim = new(300,300);

    public async Task<TEntity> InsertAsync<TEntity>(TEntity entity) where TEntity : BaseEntity
    {
        // ✗ ELIMINAR: await _semaphoreSlim.WaitAsync();
        IMongoCollection<TEntity> collection = _mongoDatabase.GetCollection<TEntity>(typeof(TEntity).Name);
        await collection.InsertOneAsync(entity);
        return entity;
        // ✗ ELIMINAR: finally block con Release()
    }
}
```

## 4. Serialización & Memoria

### 4.1 Mezcla de System.Text.Json y Newtonsoft.Json

Severidad:  Media

#### Ubicación:

- `Dev_Resources/WebApi/Extensions/ServiceExtensions.cs` línea 173
- `Dev_Resources/Persistence/Mongo/Repository.cs` línea 88
- `Dev_Resources/Application/Services/InfoProductsService.cs` líneas 80-81
- `Dev_Resources/Application/Validator/FieldValidator.cs` línea 6

#### Impacto:

- Dos librerías de serialización cargadas en memoria: +2.5 MB en runtime.
- Newtonsoft.Json se usa en:
  - HealthCheck endpoint: ~1 llamada por minuto (ServiceExtensions.cs:173)
  - Deserialización de parámetros OnBoarding: ejecutado 1 vez al inicio en constructor de InfoProductsService (líneas 80-81)
  - Manejo de errores en Repository: cada vez que hay NotFoundException (Repository.cs:88)
  - Validación de campos: en cada validación con FieldValidator (FieldValidator.cs:6)
- Los controllers principales (requests/responses HTTP) ya usan System.Text.Json correctamente.
- System.Text.Json es 10-15% más rápido que Newtonsoft.Json en las operaciones donde aún se usa.

#### Remediación:

##### 1. ServiceExtensions.cs:173 - HealthCheck middleware

#### Problema:

```
using Newtonsoft.Json;

await context.Response.WriteAsync(
    JsonConvert.SerializeObject(response)
).ConfigureAwait(false);
```

#### Solución:

```
using System.Text.Json;

await context.Response.WriteAsync(
    JsonSerializer.Serialize(response, new JsonSerializerOptions
    {
        PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
        WriteIndented = false
    })
);
```

## 2. Repository.cs:88 - Manejo de errores

### ⚠ Problema:

```
Newtonsoft.Json.JsonConvert.SerializeObject(  
    new List<ErrorDetail>() {new() {  
        Path = id,  
        Message = $"{nameEntity.Substring(0, nameEntity.Length - 6)} not valid"  
    }}  
)
```

### ✅ Solución:

```
System.Text.Json.JsonSerializer.Serialize(  
    new List<ErrorDetail>() {new() {  
        Path = id,  
        Message = $"{nameEntity.Substring(0, nameEntity.Length - 6)} not valid"  
    }},  
    new JsonSerializerOptions { PropertyNamingPolicy = JsonNamingPolicy.CamelCase }  
)
```

## 3. InfoProductsService.cs:80-81 - Deserialización de parámetros

### ⚠ Problema:

```
using Newtonsoft.Json;  
  
OnBoardingPartnerBins = JsonConvert.DeserializeObject<ProductoIdDto>(franchiseeBins!);  
LoanOnBoardingPartnerBins = JsonConvert.DeserializeObject<LoanBusinessParamDto>(loanBins!);
```

### ✅ Solución:

```
using System.Text.Json;  
  
OnBoardingPartnerBins = JsonSerializer.Deserialize<ProductoIdDto>(franchiseeBins,  
    new JsonSerializerOptions { PropertyNameCaseInsensitive = true });  
  
LoanOnBoardingPartnerBins = JsonSerializer.Deserialize<LoanBusinessParamDto>(loanBins,  
    new JsonSerializerOptions { PropertyNameCaseInsensitive = true });
```

#### 4. FieldValidator.cs:6 - Validación

##### ⚠ Problema:

```
using Newtonsoft.Json;
```

##### ✅ Solución:

```
using System.Text.Json;  
// Reemplazar todas las referencias a JsonConvert por JsonSerializer
```

#### 5. Remover paquete NuGet


```
# Ejecutar desde la raíz Dev_Resources  
dotnet remove WebApi/WebApi.csproj package Newtonsoft.Json  
dotnet remove Application/Application.csproj package Newtonsoft.Json  
dotnet remove Persistence/Persistence.csproj package Newtonsoft.Json  
dotnet remove SharedKernel/SharedKernel.csproj package Newtonsoft.Json
```

##### Beneficio esperado:

- ✅ -2.5 MB en tamaño de deployment
- ✅ -2.5 MB memoria runtime
- ✅ +10-15% throughput en operaciones de serialización afectadas
- ✅ Consistencia en naming policies
- ✅ Una sola librería de serialización

## 5. Paralelismo & DI

### 5.1 Múltiples implementaciones de misma interfaz sin keyed services

Severidad:  Alta

Ubicación:

- `Dev_Resources/WebApi/Program.cs` líneas 48-55

Impacto:

- **Inyección ineficiente:** Los controladores reciben `IEnumerable<ITokenizationCreatorService>` y deben buscar manualmente la implementación correcta usando `FirstOrDefault(x => x is CardCreatorService)`, lo cual es un anti-patrón.
- **Resolución innecesaria:** El contenedor de DI instancia TODAS las implementaciones (Card, Customer, Wallet, Loan) en cada request, aunque solo se necesita una. Esto desperdicia CPU en la construcción de objetos no utilizados y memoria heap para mantener referencias innecesarias.
- **Ejemplo:** Un endpoint de tarjetas instancia 4 servicios pero solo usa `CardCreatorService`, descartando los otros 3.
- **Código frágil:** El patrón `x is CardCreatorService` usa reflexión en runtime y es propenso a errores si se renombran las clases.
- **Solución moderna disponible:** .NET 8 introdujo *keyed services* específicamente para resolver este escenario, permitiendo inyección directa y type-safe de la implementación correcta sin overhead ni búsquedas manuales.

Remediación:

⚠ Problema:

```
// Program.cs - Todas las implementaciones registradas con misma interfaz
builder.Services.AddScoped<ITokenizationCreatorService, CardCreatorService>();
builder.Services.AddScoped<ITokenizationCreatorService, CustomerCreatorService>();
builder.Services.AddScoped<ITokenizationCreatorService, WalletCreatorService>();
builder.Services.AddScoped<ITokenizationCreatorService, LoanCreatorService>();

// Controller - Resuelve TODAS y busca la correcta
public CardTokensController(IEnumerable<ITokenizationCreatorService> services)
{
    _cardService = services.FirstOrDefault(x => x is CardCreatorService);
}
```

✅ Solución:

```
// Program.cs - Usar keyed services (.NET 8)
builder.Services.AddKeyedScoped<ITokenizationCreatorService, CardCreatorService>("card");
builder.Services.AddKeyedScoped<ITokenizationCreatorService, CustomerCreatorService>("customer");
builder.Services.AddKeyedScoped<ITokenizationCreatorService, WalletCreatorService>("wallet");
builder.Services.AddKeyedScoped<ITokenizationCreatorService, LoanCreatorService>("loan");

// Controller - Inyectar solo el servicio necesario
public CardTokensController(
    [FromKeyedServices("card")] ITokenizationCreatorService cardService)
{
    _cardService = cardService;
}
```

## 5.2 Ausencia de paralelismo en llamadas independientes

Severidad: ● Baja

### Evidencia:

- No hay uso de `Task.WhenAll` para operaciones I/O paralelas.
- Ejemplo: `CardCreatorService` obtiene product info secuencialmente.

### Remediación:

#### ✓ Solución:

```
// Si múltiples operaciones I/O son independientes:  
var productTask = _infoProductsService.GetInfoCardBinAsync(bin, tppId);  
var customerTask = _customerService.ValidateAsync(customerId);  
await Task.WhenAll(productTask, customerTask);
```

### 5.3 Constructor de `InfoProductsService` ejecuta I/O síncrono

Severidad:  Media

#### Ubicación:

- `Dev_Resources/Application/Services/InfoProductsService.cs` línea 29

#### Impacto:

- El constructor de `InfoProductsService` llama a `GetOnBoardingParams()` que hace una llamada síncrona
- La primera vez que se inyecta `InfoProductsService` en un request, ese request se bloquea esperando que termine la carga de parámetros.
- Los constructores no deben hacer operaciones I/O (regla de .NET dependency injection).

#### Remediación:

##### Problema:

```
// InfoProductsService.cs
public InfoProductsService(ILogger<InfoProductsService> logger, IGlobalParametersService globalParametersService)
{
    _logger = logger;
    _globalParametersService = globalParametersService;
    GetOnBoardingParams(); // I/O síncrono en constructor
}
```

##### Solución:

```
// Crear IHostedService que precargue parámetros antes de aceptar tráfico
public class ParametersWarmupService : IHostedService
{
    private readonly IGlobalParametersService _parametersService;

    public ParametersWarmupService(IGlobalParametersService parametersService)
    {
        _parametersService = parametersService;
    }

    public async Task StartAsync(CancellationToken cancellationToken)
    {
        await _parametersService.GetAllParametersAsync();
    }

    public Task StopAsync(CancellationToken cancellationToken) => Task.CompletedTask;
}

// Program.cs
builder.Services.AddHostedService<ParametersWarmupService>();
```

## 6. Estado Global

### 6.1 `TokenService.Token` sin thread-safety

Severidad:  Crítica


Ubicación:

- `Dev_Resources/Application/Settings/TokenService.cs` línea 10

Impacto:

- Acceso concurrente sin sincronización → data races.
- `RestClientService` lee/escribe sin locks.

Remediación:

 Problema:

```
// TokenService.cs
public static class TokenService
{
    public static Dictionary<string, TokenResponseDto??> Token { get; set; }
}
```

 Solución:

```
// TokenService.cs
public static class TokenService
{
    public static ConcurrentDictionary<string, TokenResponseDto?> Token { get; } = new();
}
```



## 6.2 `TraceIdentifier` - Componente Redundante

Severidad:  Crítica

### Ubicación:







- `Dev_Resources/Domain/Common/TraceIdentifier.cs`
- `Dev_Resources/WebApi/Extensions/ExceptionHandlerMiddleware.cs` líneas 28, 31, 35, 44
- `Dev_Resources/WebApi/Handler/EventTypeEnricher.cs` líneas 36, 37
- `Dev_Resources/WebApi/Program.cs` línea 57

### Impacto:

- **Memory leak:** El diccionario estático `traceIds` crece indefinidamente sin limpieza efectiva, causando un memory leak lento pero constante.
- **Complejidad innecesaria:** Mapea `HttpContext.TraceIdentifier` a un GUID personalizado, agregando overhead en cada request.
- **Redundante:** ASP.NET Core ya proporciona `HttpContext.TraceIdentifier` que es único por request, thread-safe, y automáticamente capturado por Serilog como `RequestId`.
- **Incompatible con standards:** El `TraceId` personalizado no sigue estándares de telemetría (W3C Trace Context) que herramientas como Instana esperan.
- **Enricher innecesario:** `EventTypeEnricher` agrega complejidad solo para mapear el `TraceId` personalizado en logs, cuando Serilog ya captura automáticamente `RequestId`.

### Análisis:

El componente `TraceIdentifier` fue creado para generar un GUID personalizado por request y mapearlo al `HttpContext.TraceIdentifier` de ASP.NET Core. Sin embargo, este enfoque es redundante porque:

-  `HttpContext.TraceIdentifier` ya es único por request
-  Ya está disponible en toda la aplicación vía `HttpContextAccessor`
-  Serilog lo captura automáticamente como `RequestId` en todos los logs
-  Es thread-safe por diseño (scope del request)
-  Sigue estándares de telemetría (W3C Trace Context)
-  Instana y otras herramientas APM lo reconocen automáticamente

Flujo actual (complejo e ineficiente):

Remediación:

⚠ Problema actual:

```
// TraceIdentifier.cs - Clase completa a eliminar
public class TraceIdentifier
{
    public string GUID { get; set; }
    private static readonly ConcurrentDictionary<string, string> traceIds = new();
    public TraceIdentifier() => GUID = ObjectId.GenerateNewId().ToString();

    public static void Add(string key, string value)
        => traceIds.TryAdd(key, value);

    public static bool TryGetValue(string key, out string value)
        => traceIds.TryGetValue(key, out value!);

    public static bool Remove(string key)
        => traceIds.TryRemove(key, out _);
}

// ErrorHandlerMiddleware.cs
public async Task Invoke(HttpContext context, TraceIdentifier traceIdentifier, ...)
{
    try
    {
        TraceIdentifier.Add(context.TraceIdentifier!, traceIdentifier.GUID!);
        await _next(context);
    }
    finally
    {
        TraceIdentifier.Remove(context.TraceIdentifier!);
    }
}
```

✅ Solución:

1. Eliminar archivos y registros:

```
# Eliminar clase TraceIdentifier
# Dev_Resources/Domain/Common/TraceIdentifier.cs

# Eliminar EventTypeEnricher
# Dev_Resources/WebApi/Handler/EventTypeEnricher.cs

# Program.cs - Eliminar registro en DI
// ❌ ELIMINAR: builder.Services.AddScoped<TraceIdentifier>();
```

## 2. Simplificar ErrorHandlerMiddleware:

### ⚠ Problema:

```
// ErrorHandlerMiddleware.cs - ANTES
public async Task Invoke(HttpContext context, TraceIdentifier traceIdentifier, ...)
{
    try
    {
        TraceIdentifier.Add(context.TraceIdentifier!, traceIdentifier.GUID!);
        await _next(context);
    }
    catch (Exception error)
    {
        await HandleException(context, traceIdentifier, error);
    }
    finally
    {
        TraceIdentifier.Remove(context.TraceIdentifier!);
    }
}
```

### ✅ Solución:

```
// ErrorHandlerMiddleware.cs - DESPUÉS
public async Task Invoke(HttpContext context, ILogger<ErrorHandlerMiddleware> logger)
{
    try
    {
        await _next(context);
    }
    catch (Exception error)
    {
        await HandleException(context, logger, error);
    }
}

private static async Task HandleException(HttpContext context, ILogger logger, Exception error)
{
    // Usar directamente HttpContext.TraceIdentifier
    var traceId = context.TraceIdentifier;

    context.Response.ContentType = "application/json";
    var responseModel = new ErrorResponse()
    {
        Id = traceId, // TraceId estándar de ASP.NET Core
        Message = error.Message
    };

    // Serilog automáticamente incluye RequestId en Logs
    logger.LogError(error, "Unhandled exception in request");

    await context.Response.WriteAsJsonAsync(responseModel);
}
```

### 3. Acceso al TraceId en servicios (si es necesario):

#### ✓ Solución:

```
// En cualquier servicio que necesite el TraceId
public class CardCreatorService
{
    private readonly IHttpContextAccessor _httpContextAccessor;
    private readonly ILogger<CardCreatorService> _logger;

    public CardCreatorService(IHttpContextAccessor httpContextAccessor, ILogger logger)
    {
        _httpContextAccessor = httpContextAccessor;
        _logger = logger;
    }

    public async Task<TokenResponse> CreateTokenAsync()
    {
        var traceId = _httpContextAccessor.HttpContext?.TraceIdentifier;
        _logger.LogInformation("Creating token for TraceId: {TraceId}", traceId);
        // Serilog automáticamente agrega RequestId a este log también
    }
}
```