ExeWatch

Stop guessing. Start knowing.

La DevGuide de ExeWatch

Versión 1.50.1

Añadir un SDK de ExeWatch a una aplicación Delphi son unas pocas líneas: el Quickstart de abajo es todo lo que hace falta. Pero en cuanto la integración está en pie y el dashboard cobra vida, llega la pregunta más difícil, la que el manual de referencia no responde: ¿qué conviene vigilar en realidad?

A esa pregunta responde esta guía, que se sitúa un nivel por encima de la referencia técnica (la página "Docs" en la consola, en https://exewatch.com/ui/docs). La referencia te dice cómo instalar, inicializar y llamar al SDK. Esta te dice para qué vale la pena llamarlo, por qué y cuándo. El hilo conductor son situaciones que reconoces al vuelo: la llamada de soporte por un crash que nadie puede reproducir, la funcionalidad que no sabes si alguien usa de verdad, la release que "se siente más lenta" pero que nadie puede demostrar.

Quickstart

Una cláusula `uses` y una línea de inicialización. Todo el resto de la guía no hace más que refinar esas dos cosas.

1. Añade el SDK a tu cláusula uses. En una app VCL, es decir la inmensa mayoría de las apps Delphi, son dos units: la segunda es la que captura las excepciones de la GUI.

```
uses
  ExeWatchSDKv1, ExeWatchSDKv1.VCL;   // app FireMonkey: ExeWatchSDKv1.FMX en lugar de
.VCL
```

De las dos units de hook pones una sola, la del framework que estés usando. Si la app no tiene GUI, por ejemplo un servicio de Windows o una utilidad de línea de comandos, no añadas ni la `.VCL` ni la `.FMX`: no hay message loop que enganchar y con `ExeWatchSDKv1` a secas te basta.

2. Inicializa una sola vez, temprano (en el `.dpr` antes de `Application.Run`, o en el evento `OnCreate` del formulario principal):

```
InitializeExeWatch('ew_win_xxxxxxxx', 'acme-corp');
```

Dos argumentos: el API key que copias de la consola y el id del customer al que pertenece esta instalación. Listo, la app está monitorizada.

Lo que te has llevado con esas dos líneas. A partir de este momento los crashes quedan registrados con su stack trace sin que escribas nada más, y las dos units se reparten el trabajo. `ExeWatchSDKv1` engancha `System.ExceptProc`, donde acaban las excepciones no controladas del código

normal. `ExeWatchSDKv1.VCL` (o `.FMX`) recoge en cambio las que se escapan dentro de un evento, un `OnClick` o un `OnCreate`: las intercepta el message loop del framework y las entrega a `Application.OnException`, y ahí termina el recorrido, a `System.ExceptProc` no llegan nunca. En una app de escritorio son la categoría de crash más frecuente, y por eso cuenta la segunda unit.

Arranca también un log automático `Application started`, la confirmación de que la integración está viva, y la información del dispositivo (sistema operativo, máquina, versión del binario) se envía y queda asociada al customer, donde aparece en "Devices" en la consola. Nada de esto viaja por el hilo de tu aplicación: los eventos van a una cola en disco y un hilo en segundo plano los envía, así que una red lenta o un servidor inalcanzable no ralentizan ni bloquean la app.

La unit de hook no te quita el handler que ya tenías: si ya habías puesto tu propio `Application.OnException`, el hook registra el log y luego lo llama; si no lo tenías, llama a `Application.ShowException`. La ventana de error que veía tu usuario sigue apareciendo igual. Y si el día de mañana te olvidas de ella en otro proyecto no te quedas a ciegas: el SDK detecta que corre en una app con GUI sin hook instalado y escribe él mismo un `Warning` con tag `exewatch`, que te encuentras en el dashboard.

IMPORTANT

El stack trace llega siempre, los números de línea no: para esos hace falta el fichero `.map`. El compilador Delphi no deja los nombres de unit, método y línea dentro del ejecutable: los escribe aparte, en un fichero `.map` junto al binario. El SDK lo busca al arrancar con el mismo nombre del ejecutable (`MyApp.exe` → `MyApp.map`) y lo usa para convertir las direcciones en frames legibles. Si no lo encuentra el crash queda registrado igual, pero el trace es una columna de direcciones peladas del tipo `[00007FF6A21C3F40]`.

Para tenerlos, una sola vez: pon Project Options, Building, Delphi Compiler, Linking, "Map file" en *Detailed* (los otros niveles, *Segments* y *Publics*, no contienen los números de línea; con *Publics* obtienes los nombres de los métodos pero no las líneas). Luego distribuye el `.map` junto al ejecutable, en la misma carpeta.

¿No puedes distribuirlo? Es una decisión legítima: el `.map` es un fichero de texto de unos cuantos megabytes que expone los nombres internos de tu código. En ese caso archiva el `.map` de cada release junto al binario que entregaste, porque las direcciones que ves en el dashboard siguen siendo resolubles más adelante, pero solo con el map de esa misma build: recompilar las mueve y el map nuevo no sirve de nada. Quien no quiera gestionar ficheros aparte tiene otros dos caminos: JclDebug, que pliega los símbolos dentro del propio ejecutable, y madExcept o EurekaLog si ya los usas, que resuelven el stack en tiempo de enlace y entregan a ExeWatch un

> trace ya simbolizado. Los encuentras los dos más adelante, en la nota de la Receta #1.

3. Registra lo que te interesa. De aquí en adelante registras, cuentas y mides tiempos cada vez que tengas algo que merezca quedar guardado:

```
EW.Info('Informe mensual generado', 'reporting');
EW.IncrementCounter('report.generated', 1, 'reporting');
```

Abre el dashboard y los eventos están ahí. De aquí en adelante la guía sirve para que se te ocurra todo lo que merece acabar dentro, porque la lista es más larga que la que se te ocurriría ahora mismo.

Después, una configuración cada vez

Nada de lo que sigue hace falta para arrancar: añádelo cuando de verdad lo necesites, una línea cada vez.

¿Quieres saber de qué release viene un evento? Pasa un tercer argumento.

```
InitializeExeWatch('ew_win_xxxxxxxx', 'acme-corp', '4.2.0');
```

Es `AppVersion`, tu etiqueta de release ('4.2.0', '2026-Q1', 'v2-beta'). La versión del binario es un campo aparte, que el SDK ya lee por su cuenta del ejecutable: este tercer argumento sirve cuando tu idea de release no coincide con el número compilado en el `.exe`.

¿Al arrancar todavía no sabes quién es el customer? Inicializa igual, con id vacío, y lo pones en cuanto lo descubras. El cómo y el porqué están en la sección *Inicialización: más allá de la única línea*, un poco más adelante.

Integrar cada SDK

El Quickstart era Delphi, el SDK insignia. El *qué* y el *por qué* del resto de la guía valen idénticos para cada SDK: solo cambia cómo integrarlo e inicializarlo. Aquí tienes la configuración de cada uno. En todos ellos el último argumento es la etiqueta de release (`AppVersion`), y el API key lleva el prefijo de la plataforma (`ew_win_`, `ew_lin_`, `ew_web_`, y así sucesivamente).

Delphi (nativo). Como en el Quickstart: `ExeWatchSDKv1` en la cláusula uses, más `ExeWatchSDKv1.VCL` (o `.FMX`) si la app tiene GUI, y luego `InitializeExeWatch(ApiKey, CustomerId, '4.2.0')`. La unit VCL/FMX captura por ti las excepciones de GUI no controladas.

.NET. Referencia el paquete `ExeWatch` (añade `ExeWatch.WinForms` para una app WinForms), y luego inicializa una vez al arrancar:

```
using ExeWatch;

EW.Initialize("ew_win_xxxxxxxx", "acme-corp", "4.2.0");
ExeWatchWinForms.Install();   // solo WinForms, antes de Application.Run()

EW.Info("Aplicación iniciada", "startup");
```

Una app de consola o de servicio se salta la línea `Install`: el cliente engancha `AppDomain.UnhandledException` por su cuenta. WinForms necesita en cambio la llamada `Install` explícita antes de `Application.Run`.

Python. Instala el SDK, inicializa el singleton y usa el objeto `ew` a nivel de módulo a partir de entonces:

```
from exewatch import initialize_exewatch, ew

initialize_exewatch("ew_win_xxxxxxxx", "acme-corp", app_version="4.2.0")

ew.info("Servicio iniciado", "startup")
ew.increment_counter("job.run", 1, "jobs")
```

Python no tiene GUI que enganchar, así que captura los fallos donde los manejes con `ew.error_with_exception(exc, "tag")`, o apunta `sys.excepthook` al SDK para tener una red global.

JavaScript (navegador). Declaras la configuración en `window.ewConfig` y luego cargas el script desde exewatch.com. El key es un key `ew_web_`, y el SDK captura automáticamente `window.onerror`:

```
<!-- primero la configuración... -->
<script>
  window.ewConfig = {
    apiKey: 'ew_web_xxxxxxxx',
    customerId: 'acme-corp',
    appVersion: '4.2.0'
  };
</script>

<!-- ...y luego el script, servido desde exewatch.com -->
<!-- Producción (minificado, 12 KB) -->
<script src="https://exewatch.com/static/js/exewatch.v1.min.js"></script>

<!-- Desarrollo (legible, 35 KB) -->
<script src="https://exewatch.com/static/js/exewatch.v1.js"></script>
```

El orden importa: el SDK se inicializa solo en el `DOMContentLoaded` leyendo `window.ewConfig`, así que la configuración tiene que estar ya en la página cuando se carga el script. El script lo sirves desde exewatch.com, no desde una copia tuya, así las correcciones llegan a tus usuarios sin que tengas que redistribuir nada.

Este SDK es solo para navegador, no hay build para Node. Los errores no capturados se recogen por ti, junto con los fallos de `fetch` y `XHR` y las `console.error`. En una página que carga scripts de terceros, `ignoreUrls` dentro de la misma `ewConfig` descarta los errores que llegan de dominios que no controlas, como publicidad y analítica: es ruido que no te dice nada sobre tu aplicación.

DLL (C, C++, VB, .NET antiguo, cualquier cosa con un C ABI). Carga `ExeWatchSDKv1DLL.dll` y llama a los exports planos. Las cadenas son wide (`PWideChar`), toda función es `stdcall`, y los resultados vuelven como códigos de retorno:

```
ew_Initialize(L"ew_win_xxxxxxxx", L"acme-corp", L"4.2.0");
ew_IncrementCounter(L"report.generated", 1.0, L"reporting");
```

Como un C ABI plano no puede recorrer la pila del host ni ejecutar un callback, dos tareas recaen sobre el host: pasar una cadena de stack ya resuelta a `ew_ErrorWithStackTrace`, y accionar los gauges periódicos desde tu propio timer (no hay callback de gauge periódico). Un host Delphi que prefiera la DLL sobre las units nativas puede usar la import unit `ExeWatchSDKv1Imports` y llamar a `EWInitialize(…)` en lugar de `InitializeExeWatch`.

Lo mismo, entero, con MSVC. Ninguna import library y ningún runtime de Embarcadero: define `EW_DYNAMIC_LOAD` antes del header y cada `ew_*` se convierte en un puntero a función, que `ExeWatchSDKv1.dynload.c` resuelve en tiempo de ejecución con `LoadLibrary` y `GetProcAddress`. Esto es un programa completo, no un fragmento:

```
#define EW_DYNAMIC_LOAD
#include "ExeWatchSDKv1.h"
#include <cstdio>

int wmain()
{
    // busca ExeWatchSDKv1DLL_x64.dll en la ruta de búsqueda estándar de Windows
    if (ew_LoadSDK() != EW_OK)
    {
        fwprintf(stderr, L"ExeWatchSDKv1DLL_x64.dll no encontrada\n");
        return 1;
    }

    if (ew_Initialize(L"ew_win_xxxxxxxx", L"acme-corp", L"4.2.0") != EW_OK)
    {
        wchar_t err[1024] = {};
```

```
        ew_GetLastError(err, 1024);
        fwprintf(stderr, L"Init fallida: %ls\n", err);
        ew_UnloadSDK();
        return 1;
    }

    ew_Info(L"Aplicación de ejemplo iniciada", L"startup");
    ew_IncrementCounter(L"report.generated", 1.0, L"reporting");

    ew_WaitForSending(15);   // devuelve cuántos eventos quedan en cola: 0 = todo
enviado
    ew_Shutdown();
    ew_UnloadSDK();
    return 0;
}
```

Se compila en un solo comando, desde un prompt "x64 Native Tools Command Prompt for VS 2022", pasando la carpeta que contiene el header y el loader:

```
cl /EHsc /W4 /nologo /I<carpeta-sdk> main.cpp <carpeta-sdk>\ExeWatchSDKv1.dynload.c
```

La única línea que aquí no es ceremonia es `ew_WaitForSending`. Una utilidad de línea de comandos puede terminar antes de que el hilo shipper haya vaciado la cola, y esa llamada escribe el buffer en disco y espera a que la cola se vacíe, devolviendo cuántos eventos siguen pendientes. No es un problema de pérdida de datos, porque la cola está en disco y arranca de nuevo en la siguiente ejecución, pero sin la espera tus eventos aparecen en el dashboard con una vuelta de retraso. El sample compilable con todo lo demás, identidad de usuario, tags globales, breadcrumbs, timings anidados y gauges, está en ExeWatchSamples/MSVCWithDLLSDK.

La misma DLL desde una consola Delphi. Un host Delphi también puede usar la DLL en lugar de las units nativas, y en dos casos es la elección correcta: cuando estás en un Delphi más antiguo que XE8, que es el mínimo del SDK nativo, y cuando tienes varias aplicaciones que actualizar distribuyendo un solo binario. La import unit `ExeWatchSDKv1Imports` llega hasta Delphi 5. Esto también es un programa entero:

```
program EWConsole;

{$APPTYPE CONSOLE}

uses
  ExeWatchSDKv1Imports;

var
  LRemaining: Integer;
begin
```

```
  if EWInitialize('ew_win_xxxxxxxx', 'acme-corp', '4.2.0') <> EW_OK then
  begin
    WriteLn('Init fallida: ', EWGetLastErrorStr);
    Exit;
  end;

  EWInfo('Proceso nocturno iniciado', 'batch');
  EWIncrementCounter('report.generated', 1.0, 'reporting');

  LRemaining := ew_WaitForSending(15);
  if LRemaining > 0 then
    WriteLn('Quedan ', LRemaining, ' eventos en cola: saldrán en la siguiente
ejecución.');

  ew_Shutdown;
end.
```

Los wrappers con el prefijo `EW` aceptan `string` y la conversión a `PWideChar` la hacen ellos, así que en tu código no aparece ningún cast. Al tratarse de una consola, valen las mismas dos líneas de antes: `ew_WaitForSending` antes de salir, y `ew_Shutdown` para cerrar limpio.

Una cosa que conviene saber antes de distribuir. Por defecto la unit enlaza la DLL de forma estática, así que `ExeWatchSDKv1DLL_x64.dll` tiene que estar junto al ejecutable en el momento del arranque: si falta, el proceso no arranca en absoluto, lo bloquea Windows con un error de DLL ausente antes incluso de tu primera línea de código. Si prefieres que la aplicación arranque igualmente y renuncie solo a la telemetría, define `EW_DYNAMIC_LOAD` en las opciones de proyecto: la unit pasa a `LoadLibrary`, y llamas a `EWLoadDLL` al arrancar comprobando su resultado, como hace el ejemplo de MSVC de arriba.

Y con Free Pascal. La DLL no tiene nada específico de Delphi: es un C ABI plano, `stdcall`, cadenas wide. En Windows la misma `ExeWatchSDKv1Imports` compila con FPC, que la unit reconoce y pone en `{$MODE DELPHI}` por sí sola. Donde `string` no es Unicode el alias interno pasa a `WideString` y los wrappers `EW*` convierten por ti, así que el código que escribes sigue siendo el del ejemplo de arriba.

Inicialización: más allá de la única línea

La llamada única a `InitializeExeWatch` del Quickstart es todo lo que la mayoría de las apps van a necesitar. Ese tercer argumento es `AppVersion`, tu etiqueta de release ('4.2.0', '2026-Q1', 'v2-beta'); la versión del binario es en cambio un campo separado, autodetectado del ejecutable, así que obtienes ambos con esa única línea. Aquí tienes a qué recurrir cuando esa línea no basta.

TIP **¿Todavía no conoces al customer? Inicializa igual.** En la mayoría de las apps reales

> el SDK debería estar activo desde la primera línea, para que un crash durante el arranque quede capturado, pero solo sabes a qué customer pertenece esta instalación después de leer un fichero de licencia o de que el usuario inicie sesión. Inicializa con un customer id vacío y ponlo en cuanto lo conozcas.

```
// en el .dpr, antes de Application.Run, para que el SDK esté activo de inmediato
InitializeExeWatch('ew_win_xxxxxxxx', '', '4.2.0');

// ... más adelante, cuando sepas quién es el customer (de un fichero de licencia o
tras el login):
EW.SetCustomerId(Session.CustomerCode);
```

Es un flujo deliberado y soportado, no un apaño. Mientras el customer id está vacío el SDK retiene el registro de device-info, porque necesita el id para asociar el dispositivo al customer correcto, y lo envía en el momento en que llamas a `SetCustomerId`. Si el id cambia más tarde (un customer distinto en la misma máquina) el SDK reenvía la device info bajo el nuevo customer, para que el dispositivo aparezca correctamente en ambos. La regla es simple: inicializa primero, identifica en cuanto puedas.

El customer y el user son dos identidades distintas, que se establecen con dos llamadas distintas. `SetCustomerId` establece el *customer*: la cuenta o el tenant al que pertenece esta instalación, que es el criterio por el que el dashboard agrupa dispositivos y alertas. Quién está usando realmente la app es otra cosa, el *user*, y eso lo estableces con `SetUser`:

```
// después de que la persona inicie sesión:
EW.SetUser('u-8842', 'mario.rossi@acme.example', 'Mario Rossi');
// al cerrar sesión:
EW.ClearUser;
```

El id, el email y el nombre viajan luego con cada evento como `user_id`, así que un crash muestra qué customer se lo encontró y también qué persona. Usa `SetCustomerId` para la cuenta y `SetUser` para la persona: son independientes, y puedes establecer uno, otro, ambos o ninguno de los dos.

WARNING

> `SetCustomerId` y `SetUser` **son ambos a nivel de proceso: encajan con una app de un solo usuario, no con un servidor multiusuario.** Cada uno es un único valor en la instancia del SDK para todo el proceso, no algo acotado a un hilo o una petición. Eso es exactamente lo correcto para una aplicación de escritorio, donde hay un customer y un usuario con la sesión iniciada a la vez. Es incorrecto para una app de servidor que atiende a muchos usuarios a la vez: dos peticiones en dos hilos sobrescribirían la identidad la una a la otra, y los eventos se atribuirían al último que la haya establecido. ExeWatch está

> pensado para aplicaciones de escritorio y de un solo usuario. En un servidor multitenant no rastrees la identidad por petición de esta manera; llévala dentro del propio evento (un tag por llamada o un campo `extra_data`).

Cuando la llamada de tres argumentos no basta, construye un `TExeWatchConfig` y pásalo en su lugar. Cada campo tiene un valor por defecto sensato, así que configura solo lo que necesites:

```
var
  Config: TExeWatchConfig;
begin
  Config := TExeWatchConfig.Create('ew_win_xxxxxxxx', 'acme-corp');
  Config.AppVersion := '4.2.0';
  Config.SampleRate := 0.25;               // envía el 25% de los eventos de rutina; Error/Fatal siempre pasan
  Config.GaugeSamplingIntervalSec := 60;   // cada cuánto lee los gauges periódicos (por defecto 30, mínimo 10)
  Config.MaxPendingAgeDays := 3;           // descarta los ficheros en cola más antiguos (por defecto 7)
  Config.AnonymizeDeviceId := True;        // sustituye por un hash el usuario en el device id (GDPR / AD)
  Config.GlobalTags := [TPair<string, string>.Create('edition', 'pro')];
  InitializeExeWatch(Config);
end;
```

Los ajustes que vale la pena conocer:

- **SampleRate** (de 0 a 1): la fracción de eventos de rutina que se envía realmente. `Error` y `Fatal` siempre se saltan el muestreo, así que puedes adelgazar el volumen de info y debug en un parque de máquinas amplio sin perder nunca un crash.
- **GlobalTags** e **InitialCustomDeviceInfo**: tags y campos de dispositivo aplicados antes incluso del primerísimo evento, para que hasta el log automático "Application started" ya lleve tu contexto.
- **GaugeSamplingIntervalSec**: cada cuánto el hilo de muestreo lee tus gauges periódicos (por defecto 30s, mínimo 10s).
- **MaxPendingAgeDays**: mientras la app está offline, los eventos se encolan en disco; los ficheros más antiguos que este valor se eliminan, para que una máquina que estuvo offline durante semanas no envíe datos rancios al reconectar (por defecto 7).
- **AnonymizeDeviceId**: reemplaza por un hash la parte del nombre de usuario del device id, para entornos GDPR o de Active Directory.
- **Endpoint**: apunta el SDK a una instancia self-hosted de ExeWatch en lugar del cloud por defecto. Solo on-premise.

Los otros SDKs adoptan los mismos conceptos bajo nombres de campo muy parecidos; la referencia tiene la firma exacta para cada uno.

Por qué existe esta guía

Un dashboard vacío impone, y la primera pregunta es siempre la misma: ¿por dónde empiezo? La respuesta corta es que si algo te interesa, hay que registrarlo. ¿Te interesa saber si esa función falla? Registra. ¿Cuántas veces se usa? Cuenta. ¿Si es lenta y cuánto? Mide. El error que se paga caro no es haber enviado algún evento de más, es encontrarte delante de un problema en producción y descubrir que justo ese trozo de código no cuenta nada.

Ese momento llega siempre, y llega un martes por la tarde con un cliente al teléfono. Tienes el stack trace del crash pero no sabes qué estaba haciendo el usuario un instante antes, porque ese paso no lo habías registrado. Sabes que la sincronización tarda una eternidad pero no qué fase se la come, porque solo habías medido el total. En esos veinte minutos nunca echas de menos las líneas que enviaste de más. Echas de menos las tres que no escribiste.

Así que empieza generoso. Si algo puede fallar, si puede volverse lento, si necesitas saber cuánto se usa, si explica por qué la app se comportó de cierta manera, mándalo. Añadirlo es fácil mientras estás escribiendo el código; añadirlo después, cuando la build en cuestión ya está instalada en trescientos clientes, significa una release y semanas de espera. Y el ruido, si el día de mañana hubiera demasiado, lo quitas cuando quieras: el nivel mínimo y el muestreo de cada aplicación se cambian desde la consola, sin recompilar nada.

Lo único que no merece la pena enviar es lo que no te dice nada ni a ti: un `Debug('estoy aquí')`, un `Info('paso 1')`, un mensaje que te parece útil mientras lo escribes solo porque en ese momento tienes el contexto en la cabeza. Cuando te lo encuentras en una lista de logs, meses después, ese contexto ya no está y la línea ha perdido todo su sentido. El resto de la guía sirve para que se te ocurra qué sí merece estar ahí, y para elegir la herramienta adecuada a cada cosa, ya que un crash, un recuento de uso y una duración se registran de tres maneras distintas.

Las cinco preguntas

Antes de las recetas, el mapa. ExeWatch te pone a disposición cinco herramientas, y cada una existe para responder una pregunta distinta. Casi toda decisión de "¿qué mido aquí?" se vuelve obvia una vez que sabes qué pregunta te estás haciendo.

La pregunta que tienes	La herramienta que la responde	A qué la apuntas
¿Qué pasó?	Logs (de debug a fatal, con un tag y stack trace)	eventos discretos que un humano querría leer: errores, cambios de estado, "el usuario hizo X"
¿Cuántos, con qué frecuencia?	Counters	cosas que cuentas y sumas: usos de funcionalidades, reintentos, exportaciones, fallos
¿Cuál es el valor ahora mismo?	Gauges	un nivel que sube y baja: memoria en MB, profundidad de cola, documentos abiertos, tamaño de caché
¿Cuánto tardó?	Timing (y traces anidados)	duraciones de operaciones, con traces para descomponer una lenta en sus fases
Avísame cuando algo va mal	Alerts	un umbral sobre el volumen de logs de error o fatal, o sobre la duración de una operación, configurado en la consola, no en código

Dos cosas son fáciles de pasar por alto, y ambas te ahorran trabajo real más adelante.

Los tags son la sexta herramienta silenciosa. Cada llamada de log, counter, gauge y timing acepta un `tag`, y existe un `SetTag` a nivel de proceso para el contexto que viaja junto a todo. Los tags son cómo recortas el dashboard en "payment" frente a "sync" frente a "ui" sin inventar una métrica separada para cada funcionalidad. Aciértalos y cada gráfico se puede filtrar exactamente como razonas tú; erra con ellos y el dashboard se convierte en ruido. La sección de higiene de tags viene a continuación, y es corta.

La device info es automática y gratis. Versión de la app, versión del binario, SO y hardware viajan con cada lote sin una sola llamada tuya. No los vuelvas a registrar. El único campo que pones a mano es `AppVersion`, la etiqueta de release, y eso es justo lo que hace posible comparar versiones más tarde.

Higiene y nomenclatura de tags

Un tag sirve para filtrar, no para transportar datos. Mantenlos cortos, pocos y estables en el tiempo, y el dashboard sigue siendo legible incluso después de unos cuantos años de añadidos.

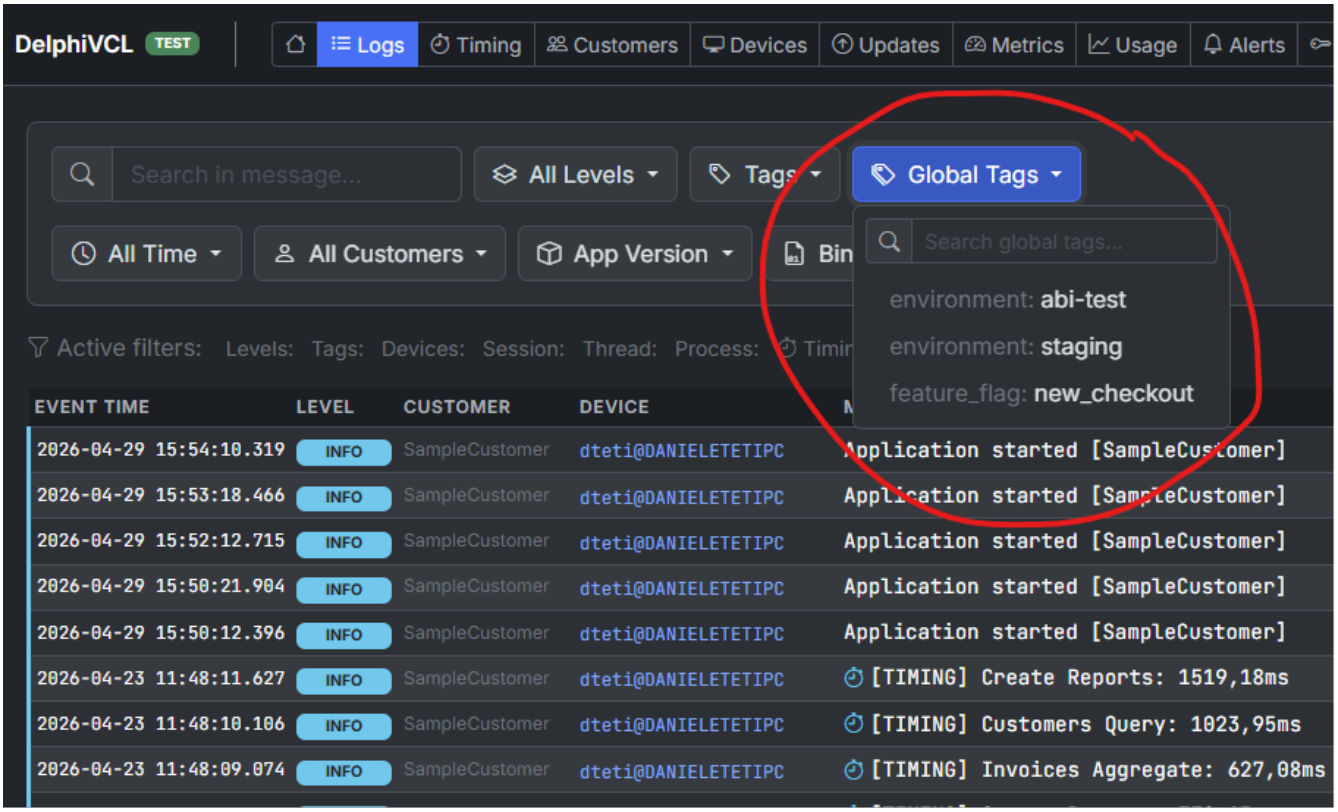
En la práctica:

- Usa un vocabulario pequeño y fijo: `payment`, `sync`, `ui`, `export`, `startup`. Un puñado de tags cubre la mayoría de las aplicaciones. Si te ves inventando un tag nuevo cada semana, para.
- Nombra los counters y los ids de timing en estilo `feature.operation`: `report.render`, `invoice.export`, `sync.retry`. Los prefijos compartidos se agrupan solos en el dashboard, así que todos tus números `sync.*` quedan juntos sin ninguna configuración.
- No metas en un tag un id, un timestamp, un nombre de fichero ni un valor que cambia en cada llamada. Cada valor distinto se convierte en una dimensión propia, así que un tag que lleve dentro el número de pedido crea miles y el filtro deja de servir para nada. Se llama alta cardinalidad. Si ese valor te hace falta, su sitio es el mensaje del log o un breadcrumb.
- Nada de datos personales en los tags ni en los mensajes de log, que se conservan y se exportan a CSV. Registra ids y resultados, no emails, nombres, tokens ni contenidos de fichero. Para la identidad está el campo de customer id.

Configura una sola vez, al arrancar, el contexto que nunca cambia, así no lo repites en cada llamada:

```
EW.SetCustomerId('acme-corp');
EW.SetTag('edition', 'pro');
EW.SetTag('channel', 'stable');
```

A partir de ahí cada evento lleva ese contexto, y puedes filtrar todo el dashboard hasta, por ejemplo, solo la edición Pro en el canal stable sin tocar otra línea de código.



Los tags que estableces con `SetTag` (o con `GlobalTags` en el init) se convierten en un filtro de "Global Tags" en la consola, así que puedes segmentar cada vista por entorno, edición, feature flag y lo que sea que hayas etiquetado.

Las mismas llamadas, en cada SDK

Las recetas que siguen usan Delphi. Los conceptos son idénticos entre SDKs; solo cambia la forma de escribirlo. Esta tabla es el mapa, para que un lector de .NET o Python pueda traducir cualquier snippet de un vistazo. La referencia de la consola tiene las firmas completas.

Lo que quieres	Delphi (nativo)	.NET (`EW`)	Python (`ew`)	JavaScript (`ew`)	DLL (C plano)
Registrar un error con stack	`EW.ErrorWithException(E, 'tag')`	`EW.ErrorWithException(ex, "tag")`	`ew.error_with_exception(exc, "tag")`	`ew.error('msg', 'tag')`	`ew_ErrorWithStackTrace(…)`
Registrar a un nivel	`EW.Info('msg', 'tag')`	`EW.Info("msg", "tag")`	`ew.info("msg", "tag")`	`ew.info('msg', 'tag')`	`ew_Log(level, …)`
Contar algo	`EW.IncrementCounter('x', 1, 'tag')`	`EW.IncrementCounter("x", 1, "tag")`	`ew.increment_counter("x", 1, "tag")`	`ew.incrementCounter('x', 1, 'tag')`	`ew_IncrementCounter(…)`

Lo que quieres	Delphi (nativo)	.NET (EW)	Python (ew)	JavaScript (ew)	DLL (C plano)
Registrar un gauge	`EW.RecordGauge('x', v, 'tag')`	`EW.RecordGauge("x", v, "tag")`	`ew.record_gauge("x", v, "tag")`	`ew.recordGauge('x', v, 'tag')`	`ew_RecordGauge(…)`
Gauge periódico	`EW.RegisterPeriodicGauge(…)`	`EW.RegisterPeriodicGauge(…)`	`ew.register_periodic_gauge(…)`	`ew.registerPeriodicGauge(…)`	*(ninguno: poll + `ew_RecordGauge`)*
Medir una operación	`EW.StartTiming` / `EndTiming`	`EW.StartTiming` / `EndTiming`	`ew.start_timing` / `end_timing`	`ew.startTiming` / `endTiming`	`ew_StartTiming` / `ew_EndTiming`
Trace anidado	`EW.StartTrace` / `EndTrace`	`EW.StartTrace` / `EndTrace`	`ew.start_trace` / `end_trace`	`ew.startTrace` / `endTrace`	`ew_StartTrace` / `ew_EndTrace`
Breadcrumb	`EW.AddBreadcrumb(…)`	`EW.AddBreadcrumb(…)`	`ew.add_breadcrumb(…)`	`ew.addBreadcrumb(…)`	`ew_AddBreadcrumb(…)`
Poner un tag	`EW.SetTag('k', 'v')`	`EW.SetTag("k", "v")`	`ew.set_tag("k", "v")`	`ew.setTag('k', 'v')`	`ew_SetTag(…)`
Poner el customer id	`EW.SetCustomerId('id')`	`EW.SetCustomerId("id")`	`ew.set_customer_id("id")`	`ew.setCustomerId('id')`	`ew_SetCustomerId(…)`
Poner el usuario actual	`EW.SetUser('id', 'email', 'name')`	`EW.SetUser("id", "email", "name")`	`ew.set_user("id", "email", "name")`	`ew.setUser({id, email})`	`ew_SetUser(…)`

Dos hechos específicos de cada SDK que vale la pena llevar a cada receta: la **DLL no tiene callback de gauge periódico**, así que el muestreo lo accionas desde tu propio timer, y **JavaScript es solo para navegador**. Todo lo demás es un renombrado uno a uno.

Recetas

Cada receta parte de una situación en la que ya has estado, o estarás, y remonta hasta la única cosa que vale la pena instrumentar. Un snippet Delphi canónico por receta; usa la tabla de arriba para traducirlo a tu SDK. Donde el comportamiento cambia de verdad, se indica.

Receta #1: Un cliente dice que crasheó, y no tienes ni idea de por qué

> El ticket de soporte dice: "el programa se cerró solo y perdí mi trabajo". Sin pasos, sin captura, sin número de versión. Sin telemetría tu única jugada es

> pedirle al cliente que reproduzca un crash que no puede reproducir a demanda, en una máquina que no puedes ver. La mitad de las veces el ticket muere ahí, sin resolver, y el bug se queda en el campo.

Esta es justo la situación para la que existen los logs y el stack trace, y la buena noticia es que activarlo es una línea de configuración. Añade `ExeWatchSDKv1.VCL` (para una app VCL) o `ExeWatchSDKv1.FMX` (para FMX) a tu cláusula uses, y el SDK engancha por ti la ruta de excepciones del framework: cada excepción de GUI no controlada se captura como `Fatal`, con tag `exception`, con su stack trace, y se descarga de inmediato para que nada se pierda mientras la app muere. Si olvidas la unit y la app es una GUI, el SDK lo nota y te avisa de que la añadas. Para cuando el cliente descuelga el teléfono, el crash ya está en tu dashboard, con el stack, la versión de la app, el SO y el customer id adjuntos. Ya no le estás pidiendo que reproduzca nada; estás leyendo lo que pasó.

Las excepciones que capturas y manejas tú mismo no viajan por ese hook, así que regístralas explícitamente con la sobrecarga de excepción, que arrastra el stack trace consigo:

```
try
  ProcessDocument(ADoc);
except
  on E: Exception do
  begin
    EW.ErrorWithException(E, 'document');
    // maneja o relanza según lo necesite tu app
  end;
end;
```

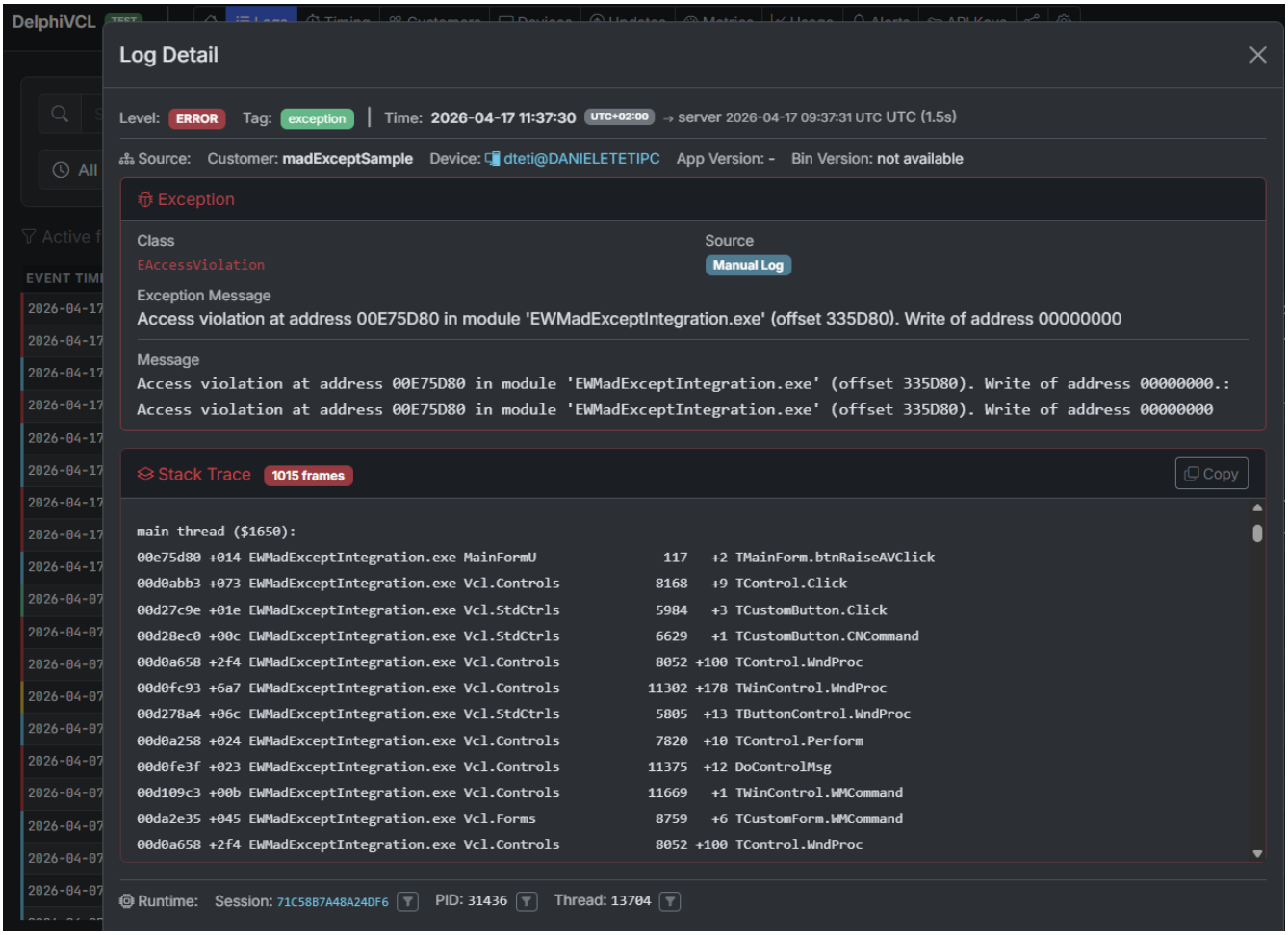
NOTE

Un stack trace solo es útil si sabes leerlo. Capturado en crudo, es una lista de direcciones de memoria, no nombres de método ni números de línea. Para obtener frames legibles tienes que entregar la info de depuración a lo que resuelva el stack. Tres formas, elige la que encaje con tu build:

- **Distribuye un fichero MAP detallado.** Pon Project Options, Linking, "Map file" en *Detailed*, y distribuye el `.map` junto al ejecutable. El SDK nativo resuelve el trace comparándolo con él.
- **Empotra los símbolos con JclDebug.** JclDebug pliega el map detallado dentro del propio binario, así que no hay nada aparte que distribuir. El SDK lee la info empotrada.
- **Reutiliza madExcept o EurekaLog si ya los tienes.** Resuelven el stack en tiempo de enlace y producen un trace totalmente simbolizado. Un puente de una sola unit pasa ese trace a ExeWatch, que conserva el stack aportado por el

> llamante en lugar de reemplazarlo. Ver https://exewatch.com/ui/docs#coexisting-madexcept.
>
> Sin ninguna de estas el crash igual queda registrado, pero el trace se queda como una secuencia de direcciones sobre las que no puedes actuar. Haz esto una vez en el momento de la release y todo reporte de crash posterior pasa a merecer la pena leerlo.

En los otros SDKs la forma es la misma. .NET es `EW.ErrorWithException(ex, "document")` y también captura en automático las excepciones no controladas. Python es `ew.error_with_exception(…)`. La DLL expone `ew_ErrorWithStackTrace(Msg, Tag, StackTrace, ExceptionClass)`: un C ABI plano no puede recorrer la pila del host, así que el host resuelve el trace y pasa la cadena. El SDK de JavaScript es solo para navegador (keys `ew_web_`) y captura en automático `window.onerror`.



Qué aspecto tiene un crash capturado en el dashboard: un `EAccessViolation` con su stack resuelto en unit, método y línea, junto a la sesión, el dispositivo y el cliente que lo sufrieron. Esta es una build con info de símbolos; sin ella los mismos frames serían direcciones peladas.

La última pieza es no tener que vigilar el dashboard en busca de crashes en absoluto. Configura una alerta sobre el recuento de `fatal` (tratado más adelante en Alertas) y un pico te llega por email, en minutos, en vez de por ticket de soporte, en días.

Receta #2: Tienes el stack, pero no lo que el usuario hizo para llegar ahí

> El error te lo sabes de memoria. Sabes hasta la línea: `SaveDocument`, una referencia nil, la misma `EAccessViolation` desde hace tres semanas. El asunto es que solo pasa en un cliente. En tu máquina no, en las otras treinta instalaciones tampoco, en la suya dos o tres veces por semana. No tienes un bug que buscar, tienes una secuencia que adivinar: algo que ese usuario hace y los demás no, que lleva el programa a un estado en el que esa línea revienta. Puedes pasarte dos semanas preguntándole "¿pero qué estaba haciendo antes?" y recibir cada vez una reconstrucción distinta, porque nadie recuerda de verdad sus propios clics. O puedes hacer que lo cuente el programa.

El stack trace responde a la pregunta *dónde*. Los breadcrumbs responden a la otra mitad, la que siempre te falta: **qué estaba haciendo el usuario antes**. Son migas que vas dejando mientras la aplicación trabaja, y ExeWatch las adjunta sola al siguiente `Error` o `Fatal`, en orden, junto al stack.

Ojo con cómo se escriben, porque es el punto en el que casi todo el mundo se equivoca la primera vez. No son cinco líneas una debajo de otra: cada una va donde la cosa ocurre de verdad, repartidas por la aplicación, y entre una y otra pasan pantallas y minutos de trabajo del usuario.

```
// TInvoiceForm.FormShow -- el usuario abre una factura
EW.AddBreadcrumb(btNavigation, 'ui', 'Abierta la factura ' + IntToStr(AInvoiceId));

// ... aquí el usuario trabaja: recorre las líneas, corrige una base imponible, guarda
...

// TUserForm.ChangeRole -- el usuario cambia de rol
EW.AddBreadcrumb(btUser, 'auth', 'Cambiado al rol admin');

// ... aquí el usuario hace más cosas, quizá durante un cuarto de hora ...

// TPriceListImport.Execute -- el usuario importa un fichero externo
EW.AddBreadcrumb(btFile, 'io',
  Format('Importado %s (%d líneas)', [ExtractFileName(AFileName), ARowCount]));

// TInvoiceDAO.ReloadLines -- ocurre por debajo, el usuario ni lo ve
EW.AddBreadcrumb(btQuery, 'db', 'Recargadas las líneas de la factura');

// TInvoiceForm.BtnExportClick -- el último paso antes del crash
```

```
EW.AddBreadcrumb(btClick, 'ui', 'Clic en Exportar');
ExportInvoice(AInvoiceId);   // <-- aquí salta la excepción, y las cinco migas la
acompañan
```

Fíjate también en los valores dentro de los mensajes: el número de la factura, el nombre del fichero, el recuento de líneas. En los tags esos valores no van nunca, porque cada valor distinto se convierte en una dimensión y te hace explotar el dashboard. En un mensaje de breadcrumb, en cambio, están perfectos, y es justo ahí donde resultan útiles, porque la diferencia entre "importado un fichero" e "importado tarifa_2026.csv con 1284 líneas" es toda la distancia entre un rastro y un diagnóstico.

Cuando ese crash llega al dashboard ya no estás leyendo que hay una access violation dentro de `SaveDocument`. Estás leyendo que ese cliente, y solo ese cliente, importa una tarifa externa antes de exportar, y que tu código de exportación nunca ha visto líneas con esa forma. La secuencia que no conseguías que te contaran está escrita ahí.

Lo mismo sirve en la versión más escurridiza del problema, aquella en la que el error le pasa a muchos clientes pero no siempre. Con una decena de crashes en el dashboard dejas de razonar por hipótesis y empiezas a comparar los rastros: si en todos aparece un cierto paso y en las sesiones sanas no aparece nunca, has terminado de adivinar. Es el mismo trabajo que harías con los logs, pero sin haber tenido que prever de antemano qué log ibas a necesitar.

El tipo es uno de un conjunto fijo de dieciséis valores (`btClick`, `btNavigation`, `btHttp`, `btQuery`, `btUser`, `btForm`, `btFile`, `btState`, `btTransaction`, `btConfig`, `btCustom` y algunos más), y le sirve al dashboard para poner un icono y hacer el rastro legible de un vistazo. También está la forma corta, `EW.AddBreadcrumb('Exportación iniciada')`, cuando te basta con dejar una nota.

Cuatro comportamientos que conviene conocer, porque cambian lo que te encuentras delante cuando hace falta:

- **El rastro es por hilo, y los hilos no se ven entre sí.** Cada hilo guarda sus propias migas. Si el usuario hace clic en Exportar en el hilo principal y el error revienta luego en un hilo en segundo plano, en el rastro adjunto a ese error el clic no está. Cuando lances un trabajo asíncrono, deja una miga también dentro del worker con lo que le has pasado, o la historia se rompe justo en el punto donde la necesitas.
- **Se conservan las últimas 20 por hilo.** La vigésimo primera tira la más antigua, así que lo que se adjunta a un crash es la carrera inmediata y no toda la sesión. Por eso no van dentro de un bucle: veinte iteraciones de `btQuery` llenan el rastro y borran el contexto que las precedía.
- **Se adjuntan solo a `Error` y `Fatal`.** Un `Info` o un `Debug` no se las lleva consigo, y por sí solas no se envían nunca. Así que no consumen quota hasta que un fallo las usa.
- **Se consumen.** Una vez adjuntadas a un error, el rastro de ese hilo se vacía, así el segundo error no arrastra la carrera del primero. Conviene saberlo: si la misma operación falla dos veces

seguidas, el segundo evento lleva solo las migas dejadas entretanto. Ante un par de errores, el que tiene la historia completa es el primero.

Qué merece la pena dejar como breadcrumb: el paso de una pantalla o un diálogo a otro (`btNavigation`, `btForm`), las llamadas externas que haces (`btHttp`), las queries que importan (`btQuery`), las acciones con las que el usuario cambia el estado del programa, login, cambio de rol, cambio de empresa (`btUser`), y los ficheros que abre o importa (`btFile`). La regla práctica es simple: si mañana, delante de un crash, te entrarían ganas de preguntar "¿pero qué había hecho antes?", esa es una miga que hay que dejar hoy.

Lo que te devuelve el dashboard: abres el crash y encuentras junto al stack los últimos veinte pasos que llevaron hasta él, en orden, con tipo y categoría. El mismo crash en dos clientes distintos se lee como dos historias distintas, y ahí es donde suele saltar cuál de las dos es la tuya.

Receta #3: Estás a punto de discutir sobre una funcionalidad que quizá nadie usa

> Una reunión de planificación. Alguien está seguro de que la funcionalidad de exportación por lotes es esencial y pide dos semanas para ampliarla. Otro cree que casi nadie la toca. Los dos están adivinando, porque ninguno tiene un número, y en este tipo de discusiones suele ganar la voz más alta. Es exactamente el tipo de pregunta que un solo counter zanja para siempre.

Deja un counter en el punto de entrada de cada funcionalidad que te importe. Aquí la herramienta correcta es un counter, no un log, porque la pregunta es "cuántos", y los counters se suman en rollups en el backend, así que lo que lees de vuelta es el total en cada instalación, barato, sin que almacenes una fila por clic.

```
EW.IncrementCounter('report.generated', 1, 'reporting');
```

TIP

Incrementa una vez por uso lógico, no en cada iteración. Si generar un informe procesa 500 líneas, sigue siendo un solo uso de la funcionalidad, así que un solo incremento, no 500. Contando las líneas medirías el tamaño de los informes, no cuántas veces la gente usa la exportación, y era la segunda la pregunta que querías responder. Si la primera también te interesa, eso es un gauge aparte: `EW.RecordGauge('report.rows', RowCount, 'reporting')`.

Un mes después el dashboard clasifica tus funcionalidades por uso real, y la discusión del roadmap

la zanjan los hechos: amplías lo que la gente usa y retiras discretamente lo que no.

Receta #4: Una release "se siente más lenta" y nadie puede demostrarlo

> Sacas la v4.2. En una semana dos clientes mencionan que el informe principal "parece flojo desde la actualización". ¿Es real, o es la sospecha habitual que sigue a cualquier cambio? No puedes saberlo, porque "se siente más lenta" no son datos, y pedirle al cliente que lo cronometre no es un plan.

La solución es medir la operación en el campo y etiquetar cada medición con la release que la produjo. Envuelve la operación en un par de timing, y pon la etiqueta de release en el init para que cada muestra sepa su versión:

```
EW.StartTiming('report.render', 'reporting');
try
  RenderReport(AReport);
finally
  EW.EndTiming('report.render');
end;
```

WARNING

Cierra siempre el par en un `try..finally` (o `try..except`). Si `RenderReport` lanza una excepción, `EndTiming` todavía tiene que ejecutarse, de lo contrario el timing queda abierto y esa muestra se pierde, justamente en las rutas de error que más quieres ver.

La etiqueta de release viene del init: `InitializeExeWatch(ApiKey, CustomerId, '4.2.0')` establece `AppVersion` para la ejecución, mientras que la versión del binario se autodetecta por separado, así que obtienes ambas. Ahora el flujo que responde la pregunta: después de que la release haya estado en el campo el tiempo suficiente para reunir muestras, abre la página "Timing", filtra por ventana temporal, y usa "Export CSV". El fichero respeta tus filtros activos y se abre en Excel, donde pivotas la duración media y p95 por `app_version`. La flojera deja de ser una opinión y se convierte en "report.render pasó de 120ms a 300ms en la 4.2", algo que detectaste con tus propios datos de campo antes de que se convirtiera en abandonos. Una vista nativa de "filtrar por versión" dentro de la interfaz está en el backlog; hasta que llegue, el pivote sobre CSV es la vía soportada, y la misma exportación funciona igual para "Logs", "Timing" y "Metrics".

Receta #5: "El checkout es lento", pero ¿lento dónde?

> Los clientes dicen que el checkout tarda una eternidad. Cronometras el conjunto y son cuatro segundos. Ese número es casi inútil por sí solo, porque ¿cuatro segundos de qué? ¿Cargar el carrito? ¿La pasarela de pago? ¿Renderizar el recibo? Optimizando a ciegas, podrías pasar un día haciendo la query de base de datos más rápida y mover el total de cuatro segundos a tres coma nueve, porque el coste real estaba en otra parte por completo.

Los traces anidados descomponen esos cuatro segundos. Inicia un trace, corre un timing alrededor de cada fase dentro de él, y cierra el trace. El backend reconstruye un waterfall, una barra por fase, para que puedas ver qué hijo domina de verdad:

```
EW.StartTrace('checkout');
try
  EW.StartTiming('checkout.db', 'checkout');
  LoadCart(ACartId);
  EW.EndTiming('checkout.db');

  EW.StartTiming('checkout.payment', 'checkout');
  ChargeCard(ACard);
  EW.EndTiming('checkout.payment');

  EW.StartTiming('checkout.render', 'checkout');
  RenderReceipt;
  EW.EndTiming('checkout.render');
finally
  EW.EndTrace;
end;
```

`StartTrace` devuelve un id de trace y `EndTrace` devuelve el total de milisegundos transcurridos, así que puedes registrar o asertar sobre el total si quieres, y cada timing anidado se convierte en un span hijo. En el dashboard "el checkout es lento" se convierte en "el pago es el 80% del checkout", lo que te dice que el problema es la pasarela, no tu código, y te ahorra el día que habrías pasado optimizando la fase equivocada. .NET y Python reflejan esto al pie de la letra; la DLL usa `ew_StartTrace(Name, Buffer, BufLen)`, que escribe el id del trace en un buffer aportado por el llamante, y `ew_EndTrace(&ElapsedMs)`.

Un trace agregado: cada fase es una barra con su avg, min, max y p95, y el porcentaje te dice cuánto del padre representa. Aquí las fases de transformación y render dominan, así que ahí es donde se va el tiempo. Las estadísticas se calculan sobre ejecuciones exitosas, así que una fase fallida no las sesga.

Qué pasa si olvidas cerrar un timing

Tarde o temprano vas a olvidar un `EndTiming`. El SDK está construido para sobrevivirlo sin fugar memoria ni reportar un número equivocado, pero lo que obtienes nunca es tan bueno como un par limpio, y esa es la verdadera razón del try/finally. Aquí tienes exactamente lo que le pasa a un timing abierto, según cómo acabe:

- **Nada, si simplemente se queda abierto.** Un timing sin su `EndTiming` correspondiente nunca emite una muestra. No se cuenta ni se promedia, simplemente está ausente. Pierdes en silencio esa medición.
- **Se auto-cierra como fallido si inicias el mismo id otra vez.** Llama a `StartTiming('report.render')` mientras un `report.render` anterior sigue abierto en el mismo hilo y el SDK cierra por ti el antiguo, marcado `success = false` y con flag `auto_closed`, con un Warning en tus logs ("auto-closed, duplicate StartTiming"). Al estar fallido, queda fuera de las estadísticas de duración; el Warning está ahí para decirte que el código tiene un agujero.
- **El más antiguo se desaloja si se acumulan demasiados.** Cada hilo guarda como máximo 100 timings abiertos. Abre el timing 101 y el más antiguo abierto se cierra a la fuerza como fallido, de nuevo con un Warning. Es la red de seguridad que impide que una fuga lenta de timings olvidados crezca sin límite.
- **Un trace limpia sus hijos.** Si el timing olvidado está anidado dentro de un `StartTrace`/`EndTrace`, `EndTrace` cierra a la fuerza cualquier span hijo que dejaras abierto, marcado como fallido, para que el waterfall siga estando completo.

El patrón en los cuatro casos: un `EndTiming` olvidado se convierte en una muestra fallida más un Warning, nunca en una duración limpia. Es la maquinaria protegiéndote, no una funcionalidad en la que apoyarte. Envuelve cada par en try/finally y nada de esto se dispara jamás.

Receta #6: La app va bien a las 9 y se arrastra a las 5

> Un cliente reporta que tu aplicación es ágil por la mañana y floja al final del día, y que un reinicio lo arregla hasta mañana. Ese comportamiento es una fuga de recursos, y es invisible para un reporte de crashes, porque no crashea nada. Simplemente se degrada, en silencio, a lo largo de horas, en una máquina que nunca ves.

Una fuga es un nivel que sube y no vuelve a bajar, que es exactamente lo que mide un gauge. No quieres esparcir llamadas de gauge por tu bucle de render; quieres el valor muestreado a intervalos. Registra un gauge periódico y el hilo de muestreo del SDK lee tu callback por ti, sin ningún timer propio:

```
EW.RegisterPeriodicGauge('mem.working_set_mb',
  function: Double
  begin
    Result := CurrentWorkingSetMB;
  end,
  'runtime');
```

Buenas cosas para vigilar así: working set en MB, recuento de GDI o de handles, recuento de documentos abiertos, tamaño de caché. Después de un día o dos el dashboard muestra la firma con claridad, un diente de sierra que sube durante toda la sesión y cae en el reinicio, y como un gauge conserva media, min, max y recuento de muestras por ventana, la media y el max crecientes son la fuga. Segmenta por `app_version` y a menudo puedes fijarla a la release exacta que la introdujo.

Una divergencia a respetar: el callback del gauge periódico existe en Delphi, .NET (un `Func<double>`), Python y JS, pero no en la DLL, porque un callback no puede cruzar el C ABI plano. Con la DLL el host es dueño del intervalo: corre tu propio timer y llama a `ew_RecordGauge(Name, Value, Tag)` en cada tick.

Receta #7: ¿Es todo el mundo, o solo un cliente?

> Tus logs muestran una ráfaga de errores de sync y tu primer instinto es que toda la flota está fallando. Antes de entrar en pánico, pregúntate si está golpeando a todos o solo a un cliente con una configuración inusual, y date cuenta de que revisar logs en crudo de 200 instalaciones nunca te lo va a decir.

Si pones el customer id en el init y etiquetas tus logs por subsistema, puedes aislar cualquier segmento con limpieza:

```
EW.SetCustomerId('acme-corp');
// ... más adelante, en la ruta de sincronización:
EW.Error('Sincronización rechazada por el servidor', 'sync');
```

Ahora filtra el dashboard a los errores de `sync` de Acme y el cuadro se aclara en segundos: es un solo cliente, detrás de un firewall corporativo, no tu código fallando en todas partes. Da un paso más y crea una alerta acotada por `customer_external_id` y tag, y te avisan de los problemas de sync de Acme en concreto, quedándose en silencio sobre los otros 200 clientes que están bien. Tasas de error por cliente y por funcionalidad, cada una alertable por su cuenta, todo saliendo de un etiquetado consistente.

Receta #8: ¿Con qué frecuencia falla en realidad?

> "El sync a veces falla" es el tipo de reporte que oculta lo único que necesitas saber: ¿cuán a menudo es a veces? Una vez a la semana es un encogimiento de hombros; un intento de cada tres es un incidente. No puedes priorizarlo hasta que veas la proporción.

La versión más simple es un counter por resultado, segmentado por un tag de resultado:

```
if TrySync then
  EW.IncrementCounter('sync.result', 1, 'success')
else
  EW.IncrementCounter('sync.result', 1, 'failure');
```

Pero si ya estás midiendo la operación, no necesitas un segundo counter, porque `EndTiming` lleva un flag `success` como tercer parámetro (por defecto `True`):

```
EW.StartTiming('sync', 'sync');
try
  DoSync;
  EW.EndTiming('sync', nil, True);   // correcto
except
  EW.EndTiming('sync', nil, False);  // fallido, pero el timing se cierra igual
  raise;
end;
```

Este flag hace algo más sutil que un counter, y vale la pena entenderlo. Un timing fallido no se

descarta: se registra, así que puedes contar los fallos y verlos en la lista de timing. Pero se deja fuera de las estadísticas de duración. La media, min, max y p95 en la página "Timing" se calculan solo a partir de ejecuciones exitosas. Importa, porque los fallos suelen ser las llamadas más lentas, una operación que se rindió tras un timeout de 30 segundos, y si esos contaran para tu latencia pensarías que tu sync se volvió más lento cuando en realidad empezó a fallar. Con el flag de success lees la latencia real de las ejecuciones que funcionaron y la tasa de fallos, desde una sola llamada.

Counter, gauge o timing: cuál

Estas tres herramientas se confunden constantemente, y confundirlas no lanza un error, registra en silencio datos que no significan nada, lo cual es peor. La distinción es simple una vez que la dices en voz alta:

Herramienta	Responde a	Cómo agrega	Qué te cuesta equivocarte
Counter	cuántos, con qué frecuencia	sumado sobre la ventana	un gauge para "número de exportaciones" tira el total a la basura
Gauge	cuál es el valor ahora mismo	media, min, max, recuento de muestras por ventana	un counter para "profundidad de cola" suma niveles en un sinsentido
Timing	cuánto tardó	distribución de duración, con traces	un counter para "cuán lento" te da un recuento, no una duración

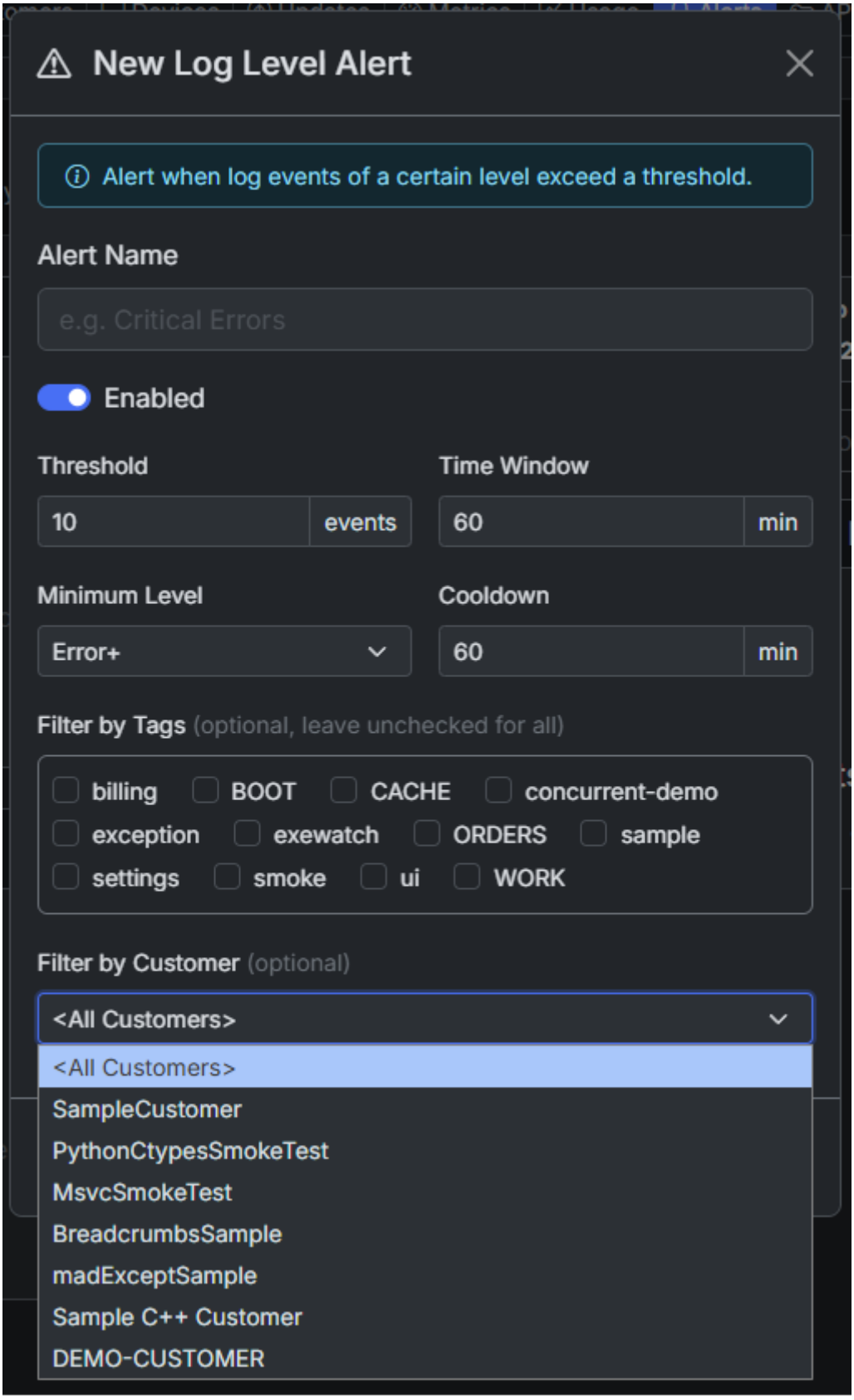
En caso de duda, prueba a sumar dos lecturas. Si la suma tiene sentido, 12 exportaciones más 8 exportaciones son 20 exportaciones, es un counter. Si no tiene sentido, una cola de 12 más una cola de 8 no es una cola de 20, es un gauge. Si lo que te importa es el tiempo transcurrido, es un timing.

Alertas que importan

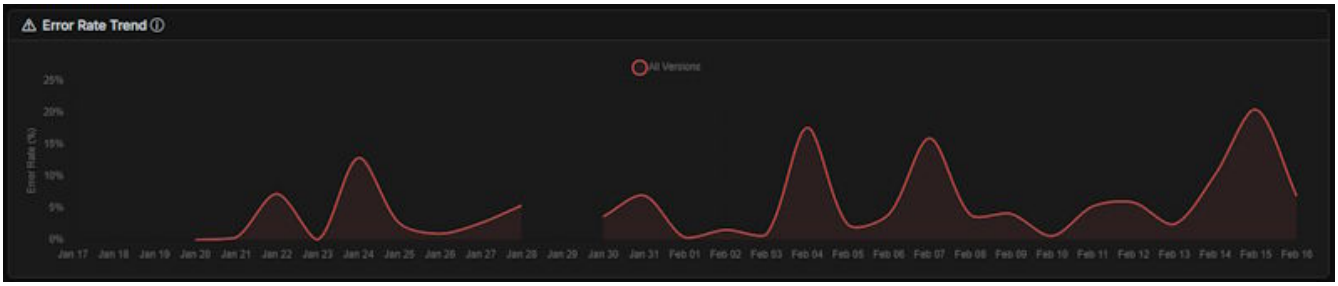
Una alerta es lo que te permite dejar de vigilar el dashboard. En vez de asomarte de vez en cuando con la esperanza de pillar un problema, describes el problema una vez y ExeWatch te envía un email cuando ocurre. Hoy existen dos tipos, y conviene saber con precisión sobre qué se dispara

cada uno, porque los límites honestos del alerting condicionan cómo usas todo lo de arriba.

Las alertas de volumen de logs se disparan cuando el recuento de eventos de log iguales o superiores a un nivel cruza un umbral dentro de una ventana. Configuras `threshold` (un recuento), `window_minutes`, `min_level` (por defecto `error`), un filtro `tags` opcional, un `customer_external_id` opcional, y `cooldown_minutes` (por defecto 60) para que un solo incidente no te avise cincuenta veces. Una típica dice: "más de 20 eventos `fatal` en 15 minutos para el cliente Acme".



La misma alerta como formulario en la consola, en "Configure alerts": un umbral sobre una ventana temporal a un nivel mínimo, con un cooldown, si hace falta estrechada por tag y por cliente. Acotar "Filter by Customer" a Acme es como consigues que te avisen de los errores de Acme sin enterarte de los de nadie más.



Tasa de error a lo largo del tiempo. Un pico como los de aquí es exactamente lo que vigila una alerta de volumen de logs, para que te llegue por email en el momento en que empieza en vez de cuando por casualidad abres este gráfico.

Las alertas de timing se disparan cuando operaciones que coinciden con un patrón de id de timing corren más lentas que un umbral de duración, con la frecuencia suficiente para cruzar un umbral de recuento dentro de una ventana. Los campos son un `timing_id_pattern`, un umbral de duración en milisegundos, un `threshold` de ocurrencias (por defecto 5), `window_minutes`, un `customer_external_id` opcional, y `cooldown_minutes` (por defecto 240). Así es como te enteras de que "report.render está corriendo lento en el campo" sin tener que vigilarlo.

NOTE

Es igual de importante saber lo que las alertas no hacen hoy. No hay una alerta de "avísame cuando un gauge suba por encima de X" ni de "avísame cuando un counter supere X". Las alertas se disparan sobre recuentos de eventos de log y sobre timing, no sobre valores arbitrarios de métrica. Así que si quieres que te avisen cuando la memoria o la profundidad de cola crucen un umbral, la respuesta honesta es que todavía no puedes; vigilas el gauge en el dashboard. El alerting por umbral sobre valores de métrica por ahora no es una funcionalidad.

La forma de mantener las alertas útiles en vez de ignoradas: mantén honestos tus niveles de log (ver anti-patrones, o tu umbral de `error` se dispara con cosas que no son errores), da a cada alerta un ámbito por tag o por cliente para que tenga un dueño claro, y pon un cooldown lo bastante largo para que un incidente real llegue como un solo email.

Comparar versiones con la exportación CSV

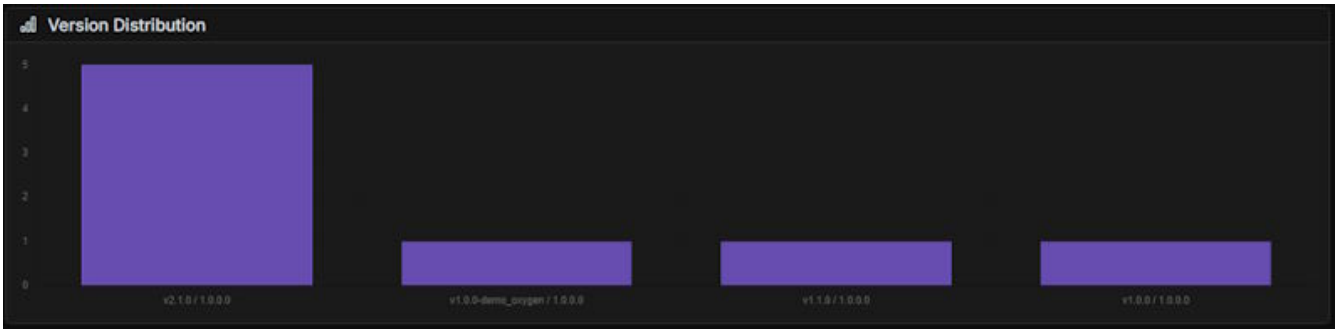
Sacas una versión y quieres saber si se volvió más lenta o más ruidosa que la anterior. Hoy esa comparación pasa por la exportación CSV, y vale la pena explicitar el flujo porque responde a una pregunta que los clientes plantean constantemente.

1. Saca la versión con `AppVersion` puesto en el init (`InitializeExeWatch(ApiKey, CustomerId, '4.2.0')`). Cada evento sabe ahora qué versión lo produjo.
2. Dale tiempo en el campo para reunir muestras, luego abre la página "Timing" y pon tus filtros:

ventana temporal, tag, cliente.

3. Usa "Export CSV". El fichero respeta esos filtros activos y se abre en Excel.
4. Crea una pivot de duración media y p95 por `app_version`, y la regresión, si la hay, está justo ahí en la tabla dinámica.

La misma exportación existe en las páginas "Logs" y "Metrics", así que puedes comparar volumen de errores o niveles de métrica entre versiones de la misma forma. Un filtro de timing por versión dentro de la interfaz está en el backlog; cuando llegue, esto pasa a ser un par de clics en vez de un pivote. Hasta entonces, la vía del CSV es la forma soportada de hacer comparación versión a versión, y funciona lo bastante bien como para que bastantes equipos se queden aquí.



Como cada evento lleva su `AppVersion`, la consola ya sabe cómo están repartidas tus releases en el campo. Esa misma etiqueta de versión es sobre la que creas la pivot del timing exportado.

Poniéndolo todo junto: instrumentar un método real

Las llamadas sueltas son fáciles de aprobar con un gesto y difíciles de ubicar. Así que aquí tienes instrumentación metida en código ordinario: un método de sync en un data module, del tipo que tiene toda app Delphi de gestión. Primero la versión desnuda, que hace su trabajo sin telemetría:

```
procedure TSyncModule.SyncInvoices;
var
  Response: IHTTPResponse;
begin
  Response := FHttp.Get(FBaseUrl + '/invoices?since=' + FLastSync);
  if Response.StatusCode <> 200 then
    raise ESyncError.CreateFmt('Sincronización rechazada por el servidor: HTTP %d',
[Response.StatusCode]);

  FConnection.StartTransaction;
  try
    ImportInvoices(Response.ContentAsString);
```

```
    FConnection.Commit;
  except
    FConnection.Rollback;
    raise;
  end;
  FLastSync := NowUtcIso;
end;
```

Ahora el mismo método instrumentado, cada añadido haciendo exactamente un trabajo:

```
procedure TSyncModule.SyncInvoices;
var
  Response: IHTTPResponse;
begin
  EW.AddBreadcrumb(btHttp, 'sync', 'GET /invoices since ' + FLastSync);  // rastro
para un posible crash
  EW.StartTiming('sync.invoices', 'sync');                               // mide la
operación entera
  try
    Response := FHttp.Get(FBaseUrl + '/invoices?since=' + FLastSync);
    if Response.StatusCode <> 200 then
      raise ESyncError.CreateFmt('Sincronización rechazada por el servidor: HTTP %d',
[Response.StatusCode]);

    EW.StartTiming('sync.import', 'sync');                               // aísla la
fase de base de datos
    FConnection.StartTransaction;
    try
      ImportInvoices(Response.ContentAsString);
      FConnection.Commit;
      EW.EndTiming('sync.import', nil, True);
    except
      FConnection.Rollback;
      EW.EndTiming('sync.import', nil, False);                           // ejecución
fallida, fuera de las estadísticas
      raise;
    end;

    FLastSync := NowUtcIso;
    EW.IncrementCounter('sync.completed', 1, 'sync');                    // uno por
cada sync correcta
    EW.EndTiming('sync.invoices', nil, True);
  except
    on E: Exception do
    begin
      EW.EndTiming('sync.invoices', nil, False);
      EW.ErrorWithException(E, 'sync');                                  // crash +
stack + breadcrumb
      raise;
    end;
```

```
  end;
end;
```

Qué fue dónde, y por qué:

- El **breadcrumb** va arriba, antes de la llamada que describe, para que si algo aguas abajo lanza una excepción, el rastro ya registre la petición que llevó ahí.
- El **timing exterior** (`sync.invoices`) envuelve toda la operación; el **timing interior** (`sync.import`) aísla la fase de base de datos, para que el waterfall separe el tiempo de red del tiempo de importación.
- El **servidor que responde mal lanza una excepción**, como cualquier otro fallo: con un estado distinto de 200 no hay nada que importar, y quien llamó tiene que saber que la sincronización no ocurrió. Fíjate en que esto no añade instrumentación, la quita: el fallo HTTP acaba en el mismo handler del final, que lo registra con su clase de excepción y los breadcrumbs, en lugar de tener una rama que se escribe encima sus propios logs y cierres de timing.
- Cada `EndTiming` está tanto en la ruta de éxito como en la de fallo, pasando `False` cuando la operación falla, para que un sync roto nunca contamine tus números de latencia.
- El **counter** incrementa una vez, solo en caso de éxito, para que `sync.completed` sea un recuento veraz de los syncs correctos.
- El **error** se registra con la excepción en el handler más externo, donde lleva el stack y el breadcrumb de arriba, y luego se relanza para que el manejo de errores de la app quede intacto.

Fíjate en la forma: la instrumentación enmarca el código real, no lo reemplaza. Borra cada línea `EW.` y el método sigue funcionando exactamente igual que antes. La instrumentación se limita a observar.

Anti-patrones

Ninguno de estos es "has registrado demasiado". Son los casos en los que lo que envías te despista en lugar de ayudarte, y la corrección está junto a cada uno.

Repetir la misma línea dentro de un bucle. Una llamada `Debug` en un bucle caliente, o un `IncrementCounter` en cada iteración. El asunto no es la cantidad: es que quinientas líneas idénticas responden todas a la misma pregunta que ya había respondido la primera, y mientras tanto consumen la quota que hacía falta para los eventos de verdad. Registra la operación, no sus iteraciones: una línea cuando arranca, otra cuando termina o falla, y un solo incremento de counter por operación lógica. Si el detalle por iteración te hace falta de verdad durante un diagnóstico,

mantenlo a nivel `Debug` y sube el nivel mínimo de la aplicación desde la consola cuando hayas acabado.

Datos personales en los logs. Emails, nombres, tokens o contenidos de fichero en un mensaje o un tag. Los logs se conservan y se exportan a CSV, y los tags son dimensiones de baja cardinalidad, no cargas útiles, así que esto es a la vez un problema de privacidad y un lío. Registra ids y categorías; usa el campo de customer id para la identidad.

Niveles que mienten. Todo registrado como `Error`, o fallos genuinos registrados como `Info`. Las alertas usan por defecto `min_level = error`, así que si tus niveles están mal tus alertas están mal con ellos, y o te avisan por nada o nunca te avisan. La escala que los mantiene honestos: Fatal significa que la app no puede continuar, Error que una operación falló, Warning degradado pero funcionando, Info un cambio de estado notable, Debug detalle solo para desarrolladores.

Gauge y counter, intercambiados. Un counter para "profundidad de cola actual" suma niveles en un total sin sentido; un gauge para "número de exportaciones" tira el total a la basura. Aplica la prueba de la suma de la tabla de arriba antes de elegir.

Tags que explotan. Un id, un timestamp, o cualquier valor por petición en un tag. Los tags son dimensiones con un conjunto pequeño y fijo de valores; los valores no acotados se multiplican en miles de dimensiones de un solo uso y dejan el dashboard inservible. Mantén el vocabulario pequeño.

Volver a registrar lo que ya tienes gratis. Registrar a mano SO, versión o hardware que ya viajan en el snapshot de dispositivo con cada lote. No lo hagas. El único campo que pones a mano es `AppVersion`, la etiqueta de release.

¿Qué pasa si se cae la conexión a internet?

Las apps de escritorio corren en portátiles que se meten en túneles, en máquinas detrás de VPNs caprichosas, en un PC que alguien desenchufa de la red a la hora de cerrar. Así que la pregunta es legítima: ¿qué le pasa a tu telemetría cuando el SDK no puede alcanzar el servidor?

No se pierde nada. El SDK no envía los eventos directamente desde el punto en el que los llamas. Los acumula en un buffer, los escribe a disco, y de enviarlos se encarga un hilo shipper en segundo plano. Cuando la red está caída el envío simplemente falla, los ficheros se quedan en disco, y el shipper reintenta a intervalos. En el momento en que vuelve la conectividad, los ficheros en cola se envían en orden. Un crash registrado en un avión aparece en tu dashboard la primera vez que el portátil vuelve a estar online.

Lo gobiernan cuatro ajustes de `TExeWatchConfig`:

- `StoragePath` es donde viven los ficheros pendientes. Por defecto es una carpeta por app; apúntala a algún sitio donde tu app siempre pueda escribir.
- `FlushIntervalMs` (por defecto 5000) es cada cuánto se escribe a disco el buffer en memoria. Más corto significa menos datos en riesgo si el proceso muere a mitad de carrera; más largo significa escrituras menos frecuentes y más grandes.
- `RetryIntervalMs` (por defecto 30000) es cada cuánto reintenta el shipper tras un fallo. Es tu cadencia de reconexión: un valor más bajo vacía antes el atraso una vez que la red ha vuelto, a costa de más intentos mientras sigue caída.
- `MaxPendingAgeDays` (por defecto 7) limita cuánto tiempo se conservan los ficheros sin enviar. Una máquina offline más tiempo que esto descarta los datos más antiguos en lugar de enviar al reconectar una avalancha de semanas. Ponlo a 0 para conservarlo todo, sin límite.

El atraso puedes vigilarlo tú mismo: `GetPendingCount` devuelve cuántos eventos siguen esperando a enviarse.

NOTE

Una conexión caída no es un `429`, aunque ambos acaben con el dato que no llega al servidor. Son opuestos. Un fallo de red es temporal, así que el SDK conserva el dato y reintenta. Un `429` significa que has superado la quota, lo cual es deliberado, así que el SDK descarta el dato en lugar de reintentar. Perder telemetría por una conexión inestable requiere un apagón más largo que `MaxPendingAgeDays`; perderla por un `429` solo requiere superar tu plan, de lo que trata la siguiente sección.

Instrumenta lo que es significativo, al nivel correcto

Un hecho técnico que conviene conocer, y llega al final de la guía a propósito: la quota mensual cuenta los eventos que nos envías, no los que quedan almacenados. Cuentan juntos los logs y las actualizaciones de métricas, es decir counters, gauges y timings, no solo los logs. Los mensajes internos del SDK, los que llevan tag `ew.system`, no entran en el recuento. Y como cuenta lo que ha llegado, borrar datos libera espacio, pero no devuelve quota.

No es una invitación a instrumentar menos. El orden correcto es el que has seguido hasta aquí: primero decides qué quieres ver, y luego, si el volumen llega a ser un tema, lo gobiernas. Las herramientas para gobernarlo existen y nadie te pide renunciar a un dato que te interesa.

- El nivel mínimo se configura por aplicación desde la consola, y el SDK lo recibe en la siguiente conexión. Puedes desarrollar con todo el `Debug` que quieras y luego dejar en producción solo `Info` y por encima, sin tocar el código y sin sacar una build nueva.

- También desde la consola, el muestreo envía solo una fracción de los eventos de rutina. `Error` y `Fatal` se lo saltan siempre, así que puedes adelgazar un log verboso en diez mil instalaciones sin perder un solo crash.
- Agrega lo que es pura repetición: un incremento por operación lógica en lugar de uno por iteración de bucle.
- Usa los tags para segmentar, así una métrica bien etiquetada te ahorra diez métricas casi idénticas.

Si estás de forma estable cerca del techo con una instrumentación que necesitas entera, no hay nada que recortar: estás siguiendo más aplicaciones o más clientes que cuando elegiste el plan, y llegado ese punto conviene pasar al siguiente nivel.

NOTE

Pasada la quota mensual el SDK recibe un `429` y empieza a descartar datos, así que conviene vigilar el uso en la consola. No para registrar menos, sino para darte cuenta a tiempo de que el plan se te ha quedado corto, en vez de descubrirlo perdiendo los eventos de un pico.

En el roadmap

Algunas cosas que la gente pide están planeadas pero todavía no lanzadas. Se listan aquí, con honestidad, para que sepas dónde están hoy los bordes y no vayas buscando una funcionalidad que todavía no está.

- **Detectar cuando una app se queda en silencio.** Un dead-man's-switch, para que te enteres del cliente cuya app dejó de dar señales, no solo de aquel cuya app lanzó un error. Hoy las alertas se disparan sobre eventos que ocurren, no sobre eventos que dejan de ocurrir, así que esto todavía no es posible. Planeado.
- **Detección de anomalías.** Aflorar en automático un "aquí hay algo raro" sobre las métricas, en vez de que tú fijes umbrales fijos. Todavía no realizado.
- **Una página de usage analytics.** Una vista dedicada a la adopción y a los patrones de uso de las funcionalidades, más allá de los counters en crudo. Todavía no realizada.
- **Una API de lectura / consulta.** Hoy las API keys son solo de ingest, así que sacar tus datos para reporting automatizado espera a esto. La exportación CSV es la vía manual mientras tanto.

Para todo lo que hoy existe, la referencia del SDK en la consola tiene las firmas exactas, las claves de configuración y los pasos de instalación. Esta guía es el qué y el porqué; esa es el cómo.