Stop guessing. Start knowing.

La DevGuide di ExeWatch

Versione 1.50.1

Aggiungere un SDK ExeWatch a un'applicazione Delphi richiede poche righe: la Quickstart qui sotto è già tutto quello che serve. Ma appena l'integrazione è in piedi e la dashboard prende vita, arriva la domanda più difficile, quella a cui il manuale di riferimento non risponde: cosa conviene davvero tenere d'occhio?

È a questa domanda che risponde la presente guida, che si colloca un gradino sopra il riferimento tecnico (la pagina "Docs" nella console, all'indirizzo https://exewatch.com/ui/docs). Il riferimento spiega come installare, inizializzare e chiamare l'SDK; qui invece si parla di cosa vale la pena chiamare, perché e quando. Il filo conduttore sono situazioni che riconosci al volo: la chiamata di assistenza per un crash che nessuno riesce a riprodurre, la funzionalità che non sai se qualcuno usa davvero, la release che "sembra più lenta" ma nessuno riesce a dimostrarlo.

Quickstart

Una clausola `uses` e una riga di inizializzazione. Tutto il resto della guida non fa che rifinire quelle due cose.

1. Aggiungi l'SDK alla tua clausola uses. In un'app VCL, cioè la stragrande maggioranza delle app Delphi, sono due unit: la seconda è quella che cattura le eccezioni della GUI.

```
uses
  ExeWatchSDKv1, ExeWatchSDKv1.VCL;   // app FireMonkey: ExeWatchSDKv1.FMX al posto della .VCL
```

Delle due unit di hook ne metti una sola, quella del framework che stai usando. Se l'app non ha una GUI, per esempio un servizio Windows o un'utility a riga di comando, non aggiungere né la `.VCL` né la `.FMX`: non c'è nessun message loop da agganciare e `ExeWatchSDKv1` da solo ti basta.

2. Inizializza una volta sola, presto (nel `.dpr` prima di `Application.Run`, oppure nell'evento `OnCreate` del form principale):

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

Due argomenti: l'API key che copi dalla console e l'id del cliente a cui appartiene questa installazione. Fatto, l'app è monitorata.

Cosa ti sei portato a casa con quelle due righe. Da questo momento i crash vengono registrati con il loro stack trace senza che tu scriva altro, e le due unit si dividono il lavoro. `ExeWatchSDKv1` aggancia `System.ExceptProc`, dove finiscono le eccezioni non gestite del codice normale.

`ExeWatchSDKv1.VCL` (oppure `.FMX`) prende invece quelle che scappano dentro un evento, un `OnClick` o un `OnCreate`: le intercetta il message loop del framework e le consegna ad `Application.OnException`, e lì il giro finisce, a `System.ExceptProc` non arrivano mai. In un'app desktop sono la categoria di crash più frequente, ed è per questo che la seconda unit conta.

Parte anche un log automatico `Application started`, la conferma che l'integrazione è viva, e le informazioni sul device (sistema operativo, macchina, versione del binario) vengono inviate e collegate al cliente, così da comparire in "Devices" nella console. Niente di tutto questo viaggia sul thread della tua applicazione: gli eventi finiscono in una coda su disco e un thread in background li spedisce, quindi una rete lenta o un server irraggiungibile non rallentano né bloccano l'app.

L'unit di hook non ti porta via il gestore che avevi: se avevi già un tuo `Application.OnException`, l'hook logga e poi lo richiama; se non ce l'avevi, chiama `Application.ShowException`. La finestra di errore che il tuo utente vedeva prima continua a comparire identica. E se un domani te ne dimentichi in un altro progetto non resti al buio: l'SDK si accorge di girare in un'app con GUI senza hook installato e scrive lui stesso un `Warning` con tag `exewatch`, che ti ritrovi in dashboard.

IMPORTANT

Lo stack trace arriva sempre, i numeri di riga no: quelli servono il file `.map`. Il compilatore Delphi non lascia nomi di unit, metodi e righe dentro l'eseguibile: li scrive a parte, in un file `.map` accanto al binario. L'SDK lo cerca all'avvio con lo stesso nome dell'eseguibile (`MyApp.exe` → `MyApp.map`) e usa quello per trasformare gli indirizzi in frame leggibili. Se non lo trova il crash viene registrato ugualmente, ma il trace è una colonna di indirizzi nudi tipo `[00007FF6A21C3F40]`.

Per averli, una volta sola: imposta Project Options, Building, Delphi Compiler, Linking, "Map file" su *Detailed* (gli altri livelli, *Segments* e *Publics*, non contengono i numeri di riga; con *Publics* ottieni i nomi dei metodi ma non le righe). Poi distribuisci il `.map` insieme all'eseguibile, nella stessa cartella.

Non puoi distribuirlo? È una scelta legittima: il `.map` è un file di testo di qualche megabyte che espone i nomi interni del tuo codice. In quel caso archivia il `.map` di ogni release insieme al binario che hai consegnato, perché gli indirizzi che vedi in dashboard restano risolvibili in un secondo momento, ma solo con la map di quella identica build: ricompilare li sposta e la map nuova non serve a niente. Chi non vuole gestire file separati ha altre due strade: JclDebug, che ripiega i simboli dentro l'eseguibile stesso, e madExcept o EurekaLog se già li usi, che risolvono lo stack in fase di link e consegnano a ExeWatch un trace già simbolizzato. Le trovi entrambe più avanti, nella nota della Ricetta #1.

3. Logga ciò che ti interessa. Da qui in poi logghi, conti e cronometri ogni volta che hai qualcosa che merita di essere registrato:

```
EW.Info('Report mensile generato', 'reporting');
EW.IncrementCounter('report.generated', 1, 'reporting');
```

Apri la dashboard e trovi gli eventi. Da qui in avanti la guida serve a farti venire in mente tutto quello che merita di finire lì dentro, perché la lista è più lunga di quella che ti verrebbe in mente adesso.

Poi, una configurazione alla volta

Niente di quello che segue serve per partire: aggiungilo quando ti serve davvero, una riga per volta.

Vuoi sapere da quale release arriva un evento? Passa un terzo argomento.

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

È `AppVersion`, la tua etichetta di release ('4.2.0', '2026-Q1', 'v2-beta'). La versione del binario è un campo a parte, che l'SDK legge già da solo dall'eseguibile: questo terzo argomento serve quando la tua idea di release non coincide con il numero compilato nel `.exe`.

All'avvio non sai ancora chi è il cliente? Inizializza lo stesso, con id vuoto, e lo imposti appena lo scopri. Come e perché è nella sezione *Inizializzazione: oltre la singola riga*, poco più avanti.

Integrare ogni SDK

La Quickstart era in Delphi, l'SDK di punta. Il *cosa* e il *perché* nel resto della guida valgono identici per ogni SDK: cambia solo il modo di integrarlo e inizializzarlo. Ecco il setup per ciascuno. In tutti l'ultimo argomento è l'etichetta di release (`AppVersion`), e l'API key ha il prefisso della piattaforma (`ew_win_`, `ew_lin_`, `ew_web_` e così via).

Delphi (nativo). Come nella Quickstart: `ExeWatchSDKv1` nella clausola uses, più `ExeWatchSDKv1.VCL` (oppure `.FMX`) se l'app ha una GUI, poi `InitializeExeWatch(ApiKey, CustomerId, '4.2.0')`. La unit VCL/FMX cattura per te le eccezioni GUI non gestite.

.NET. Referenzia il pacchetto `ExeWatch` (aggiungi `ExeWatch.WinForms` per un'app WinForms), poi inizializza una volta all'avvio:

```
using ExeWatch;
```

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

EW.Info("Applicazione avviata", "startup");
```

Un'app console o un servizio salta la riga `Install`: il client aggancia `AppDomain.UnhandledException` da solo. WinForms richiede invece la chiamata `Install` esplicita prima di `Application.Run`.

Python. Installa l'SDK, inizializza il singleton e poi usa l'oggetto `ew` a livello di modulo:

```
from exewatch import initialize_exewatch, ew

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

ew.info("Servizio avviato", "startup")
ew.increment_counter("job.run", 1, "jobs")
```

Python non ha una GUI da agganciare, quindi cattura i fallimenti dove li gestisci con `ew.error_with_exception(exc, "tag")`, oppure punta `sys.excepthook` sull'SDK per avere una rete globale.

JavaScript (browser). Dichiari la configurazione in `window.ewConfig` e poi carichi lo script da exewatch.com. La chiave è una chiave `ew_web_`, e l'SDK cattura automaticamente `window.onerror`:

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

<!-- ...poi lo script, servito da exewatch.com -->
<!-- Produzione (minificata, 12 KB) -->
<script src="https://exewatch.com/static/js/exewatch.v1.min.js"></script>

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

L'ordine conta: l'SDK si inizializza da solo al `DOMContentLoaded` leggendo `window.ewConfig`, quindi la configurazione deve essere già sulla pagina quando lo script viene caricato. Lo script lo servi da exewatch.com, non da una copia tua, così le correzioni arrivano ai tuoi utenti senza che tu debba ridistribuire niente.

Questo SDK è solo per browser, non esiste una build per Node. Gli errori non catturati vengono raccolti per te, insieme ai fallimenti di `fetch` e `XHR` e alle `console.error`. In una pagina che carica script di terze parti, `ignoreUrls` nella stessa `ewConfig` scarta gli errori che arrivano da domini che non controlli, tipo advertising e analytics: è rumore che non ti dice niente sulla tua applicazione.

DLL (C, C++, VB, .NET legacy, qualsiasi cosa con una ABI C). Carica `ExeWatchSDKv1DLL.dll` e chiama gli export flat. Le stringhe sono wide (`PWideChar`), ogni funzione è `stdcall` e i risultati tornano come codici di ritorno:

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

Poiché una ABI C flat non può camminare lo stack dell'host né eseguire una callback, due compiti ricadono sull'host: passare una stringa di stack già risolta a `ew_ErrorWithStackTrace`, e pilotare i gauge periodici da un timer tuo (non c'è callback per i gauge periodici). Un host Delphi che preferisce la DLL alle unit native può usare la import unit `ExeWatchSDKv1Imports` e chiamare `EWInitialize(…)` al posto di `InitializeExeWatch`.

Lo stesso, per intero, con MSVC. Nessuna import library e nessun runtime Embarcadero: definisci `EW_DYNAMIC_LOAD` prima dell'header e ogni `ew_*` diventa un puntatore a funzione, che `ExeWatchSDKv1.dynload.c` risolve a runtime con `LoadLibrary` e `GetProcAddress`. Questo è un programma completo, non un frammento:

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

int wmain()
{
    // cerca ExeWatchSDKv1DLL_x64.dll nel path di ricerca standard di Windows
    if (ew_LoadSDK() != EW_OK)
    {
        fwprintf(stderr, L"ExeWatchSDKv1DLL_x64.dll non trovata\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 fallita: %ls\n", err);
        ew_UnloadSDK();
        return 1;
    }
```

```
    ew_Info(L"Applicazione di esempio avviata", L"startup");
    ew_IncrementCounter(L"report.generated", 1.0, L"reporting");

    ew_WaitForSending(15);   // restituisce quanti eventi restano in coda: 0 = tutto
spedito
    ew_Shutdown();
    ew_UnloadSDK();
    return 0;
}
```

Si compila in un comando solo, da un prompt "x64 Native Tools Command Prompt for VS 2022", passando la cartella che contiene header e loader:

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

L'unica riga che qui non è cerimonia è `ew_WaitForSending`. Un'utility a riga di comando può terminare prima che il thread shipper abbia svuotato la coda, e quella chiamata scrive il buffer su disco e aspetta che la coda si svuoti, restituendo quanti eventi sono ancora in attesa. Non è un problema di perdita di dati, perché la coda è su disco e riparte alla prossima esecuzione, ma senza l'attesa i tuoi eventi compaiono in dashboard con un giro di ritardo. Il sample compilabile con tutto il resto, identità utente, tag globali, breadcrumb, timing annidate e gauge, è in ExeWatchSamples/MSVCWithDLLSDK.

La stessa DLL da una console Delphi. Anche un host Delphi può usare la DLL invece delle unit native, e in due casi è la scelta giusta: quando sei su un Delphi più vecchio di XE8, che è il minimo del SDK nativo, e quando hai più applicazioni che devi aggiornare distribuendo un binario solo. La import unit `ExeWatchSDKv1Imports` arriva fino a Delphi 5. Anche questo è un programma intero:

```
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 fallita: ', EWGetLastErrorStr);
    Exit;
  end;

  EWInfo('Batch notturno avviato', 'batch');
```

```
  EWIncrementCounter('report.generated', 1.0, 'reporting');

  LRemaining := ew_WaitForSending(15);
  if LRemaining > 0 then
    WriteLn('Restano ', LRemaining, ' eventi in coda: partiranno alla prossima
esecuzione.');

  ew_Shutdown;
end.
```

I wrapper con il prefisso `EW` accettano `string` e la conversione a `PWideChar` la fanno loro, quindi nel tuo codice non compare un cast. Trattandosi di una console, valgono le stesse due righe di prima: `ew_WaitForSending` prima di uscire, e `ew_Shutdown` per chiudere pulito.

Una cosa da sapere prima di distribuire. Di default la unit lega la DLL staticamente, quindi `ExeWatchSDKv1DLL_x64.dll` deve essere accanto all'eseguibile al momento dell'avvio: se manca, il processo non parte affatto, lo blocca Windows con un errore di DLL mancante prima ancora della tua prima riga di codice. Se preferisci che l'applicazione parta comunque e rinunci soltanto alla telemetria, definisci `EW_DYNAMIC_LOAD` nelle opzioni di progetto: la unit passa a `LoadLibrary`, e chiami `EWLoadDLL` all'avvio verificandone il risultato, come fa l'esempio MSVC qui sopra.

E con Free Pascal. La DLL non ha niente di specifico di Delphi: è una ABI C flat, `stdcall`, stringhe wide. Su Windows la stessa `ExeWatchSDKv1Imports` compila con FPC, che la unit riconosce e mette in `{$MODE DELPHI}` da sé. Dove `string` non è Unicode l'alias interno diventa `WideString` e i wrapper `EW*` convertono per te, quindi il codice che scrivi resta quello dell'esempio qui sopra.

Inizializzazione: oltre la singola riga

La singola chiamata a `InitializeExeWatch` della Quickstart è tutto ciò di cui la maggior parte delle app avrà mai bisogno. Quel terzo argomento è `AppVersion`, la tua etichetta di release ('4.2.0', '2026-Q1', 'v2-beta'); la versione del binario è invece un campo separato, rilevato in automatico dall'eseguibile, quindi da quell'unica riga ottieni entrambe. Ecco a cosa ricorrere quando quella riga non basta.

TIP

Non sai ancora chi è il cliente? Inizializza lo stesso. Nella maggior parte delle app reali l'SDK dovrebbe essere attivo dalla prima riga, così che un crash durante l'avvio venga catturato, ma scopri a quale cliente appartiene l'installazione solo dopo aver letto un file di licenza o dopo che l'utente ha fatto login. Inizializza con un id cliente vuoto e impostalo appena lo conosci.

```
// nel .dpr, prima di Application.Run, così l'SDK è attivo da subito
```

```
InitializeExeWatch('ew_win_xxxxxxxx', '', '4.2.0');

// ... più avanti, quando sai chi è il cliente (da un file di licenza o dopo il
login):
EW.SetCustomerId(Session.CustomerCode);
```

È un flusso deliberato e supportato, non un espediente. Finché l'id cliente è vuoto l'SDK trattiene il record di device-info, perché gli serve l'id per collegare il device al cliente giusto, e lo invia nell'istante in cui chiami `SetCustomerId`. Se in seguito l'id cambia (un cliente diverso sulla stessa macchina) l'SDK reinvia la device-info sotto il nuovo cliente, così il device compare correttamente per entrambi. La regola è semplice: prima inizializza, poi identifica appena puoi.

Cliente e utente sono due identità diverse, impostate con due chiamate diverse. `SetCustomerId` imposta il *cliente*: l'account o tenant a cui appartiene questa installazione, ossia il criterio con cui la dashboard raggruppa device e alert. Chi sta effettivamente usando l'app è un'altra cosa, l'*utente*, e quello lo imposti con `SetUser`:

```
// dopo che la persona ha fatto login:
EW.SetUser('u-8842', 'mario.rossi@acme.example', 'Mario Rossi');
// al logout:
EW.ClearUser;
```

Id, email e nome viaggiano poi con ogni evento come `user_id`, così un crash mostra non solo quale cliente lo ha incontrato ma anche quale persona. Usa `SetCustomerId` per l'account e `SetUser` per la persona: sono indipendenti, e puoi impostare l'uno, l'altro, entrambi o nessuno dei due.

WARNING

`SetCustomerId` **e** `SetUser` **sono entrambi validi per l'intero processo: vanno bene per un'app single-user, non per un server multi-utente.** Ciascuno è un singolo valore sull'istanza dell'SDK per tutto il processo, non qualcosa legato a un thread o a una richiesta. È esattamente ciò che serve a un'applicazione desktop, dove un solo cliente e un solo utente autenticato sono attivi alla volta. È sbagliato invece per un'app lato server che serve molti utenti insieme: due richieste su due thread si sovrascriverebbero a vicenda l'identità, e gli eventi verrebbero attribuiti a chi l'ha impostata per ultimo. ExeWatch è pensato per applicazioni desktop e single-user. Su un server multi-tenant non tracciare l'identità per-richiesta in questo modo; portala dentro l'evento stesso (un tag per-chiamata o un campo `extra_data`).

Quando la chiamata a tre argomenti non basta, costruisci un `TExeWatchConfig` e passa quello. Ogni campo ha un default sensato, quindi imposta solo ciò che ti serve:

```
var
```

```
  Config: TExeWatchConfig;
begin
  Config := TExeWatchConfig.Create('ew_win_xxxxxxxx', 'acme-corp');
  Config.AppVersion := '4.2.0';
  Config.SampleRate := 0.25;                // invia il 25% degli eventi di routine;
Error/Fatal passano sempre
  Config.GaugeSamplingIntervalSec := 60;    // ogni quanto legge i gauge periodici
(default 30, minimo 10)
  Config.MaxPendingAgeDays := 3;            // scarta i file in coda più vecchi di
così (default 7)
  Config.AnonymizeDeviceId := True;         // sostituisce con un hash lo username
nell'id device (GDPR / AD)
  Config.GlobalTags := [TPair<string, string>.Create('edition', 'pro')];
  InitializeExeWatch(Config);
end;
```

Le impostazioni che vale la pena conoscere:

- **SampleRate** (da 0 a 1): la frazione di eventi di routine effettivamente inviati. `Error` e `Fatal` scavalcano sempre il campionamento, così puoi diradare il volume di info e debug su un parco macchine ampio senza mai perdere un crash.
- **GlobalTags** e **InitialCustomDeviceInfo**: tag e campi device applicati prima ancora del primissimo evento, così che perfino il log automatico "Application started" porti già con sé il tuo contesto.
- **GaugeSamplingIntervalSec**: ogni quanto il thread di campionamento legge i tuoi gauge periodici (default 30s, minimo 10s).
- **MaxPendingAgeDays**: mentre l'app è offline, gli eventi si accodano su disco; i file più vecchi di questo valore vengono eliminati, così una macchina rimasta offline per settimane non spedisce dati stantii alla riconnessione (default 7).
- **AnonymizeDeviceId**: sostituisce con un hash la parte username dell'id device, per ambienti GDPR o Active Directory.
- **Endpoint**: fa puntare l'SDK a un'istanza ExeWatch self-hosted invece del cloud di default. Solo on-premise.

Gli altri SDK adottano gli stessi concetti sotto nomi di campo molto simili; il riferimento ha la firma esatta per ognuno.

Perché esiste questa guida

Una dashboard vuota mette in soggezione, e la prima domanda è sempre la stessa: da dove comincio? La risposta breve è che se una cosa ti interessa, va registrata. Ti interessa sapere se

quella funzione fallisce? Logga. Quante volte viene usata? Conta. Se è lenta e quanto? Cronometra. L'errore che si paga caro non è aver mandato qualche evento di troppo, è ritrovarsi davanti a un problema in produzione e scoprire che proprio quel pezzo di codice non racconta niente.

Quel momento arriva sempre, e arriva di martedì pomeriggio con un cliente al telefono. Hai lo stack trace del crash ma non sai cosa stesse facendo l'utente un attimo prima, perché quel passaggio non lo avevi loggato. Sai che la sincronizzazione ci mette una vita ma non quale fase la mangia, perché avevi cronometrato solo il totale. In quei venti minuti non rimpiangi mai le righe che hai mandato in più. Rimpiangi le tre che non hai scritto.

Quindi parti generoso. Se qualcosa può fallire, se può diventare lenta, se ti serve sapere quanto viene usata, se spiega perché l'app si è comportata in un certo modo, mandala. Aggiungerne è facile finché stai scrivendo il codice; aggiungerne dopo, quando la build incriminata è già installata da trecento clienti, vuol dire un rilascio e settimane di attesa. E il rumore, se un domani ce ne fosse troppo, lo togli quando vuoi: il livello minimo e il campionamento di ogni applicazione si cambiano dalla console, senza ricompilare niente.

L'unica cosa che non vale la pena mandare è quella che non dice niente nemmeno a te: un `Debug('sono qui')`, un `Info('step 1')`, un messaggio che ti sembra utile mentre lo scrivi solo perché in quel momento il contesto ce l'hai in testa. Quando lo ritrovi in un elenco di log, mesi dopo, quel contesto non c'è più e la riga ha perso ogni scopo. Il resto della guida serve a farti venire in mente cosa invece merita di esserci, e a scegliere lo strumento adatto a ciascuna cosa, visto che un crash, un conteggio d'uso e una durata si registrano in tre modi diversi.

Le cinque domande

Prima delle ricette, la mappa. ExeWatch ti mette a disposizione cinque strumenti, e ognuno esiste per rispondere a una domanda diversa. Quasi ogni decisione del tipo "cosa traccio qui?" diventa ovvia una volta che sai quale domanda ti stai ponendo.

La domanda che hai	Lo strumento che vi risponde	Su cosa lo punti
Cosa è successo?	Log (da debug a fatal, con tag e stack trace)	eventi discreti che una persona vorrebbe leggere: errori, cambi di stato, "l'utente ha fatto X"
Quanti, quanto spesso?	Counter	cose che conti e sommi: uso di funzionalità, retry, export, fallimenti

La domanda che hai	Lo strumento che vi risponde	Su cosa lo punti
Qual è il valore in questo momento?	Gauge	un livello che sale e scende: memoria in MB, profondità di coda, documenti aperti, dimensione cache
Quanto ci ha messo?	Timing (e trace annidate)	durate di operazioni, con trace per scomporre una lenta nelle sue fasi
Avvisami quando qualcosa non va	Alerts	una soglia sul volume di log error o fatal, o sulla durata di un'operazione, impostata nella console, non nel codice

Due cose sfuggono facilmente, ed entrambe ti risparmiano lavoro vero più avanti.

I tag sono il sesto strumento silenzioso. Ogni chiamata di log, counter, gauge e timing accetta un `tag`, ed esiste un `SetTag` valido per l'intero processo per il contesto che viaggia con tutto. I tag sono il modo in cui affetti la dashboard in "payment" contro "sync" contro "ui" senza inventare una metrica separata per ogni funzionalità. Azzeccali e ogni grafico è filtrabile esattamente come ragioni tu; sbagliali e la dashboard diventa rumore. La sezione sull'igiene dei tag è la prossima, ed è breve.

La device-info è automatica e gratis. Versione app, versione binario, OS e hardware viaggiano con ogni batch senza una singola chiamata da parte tua. Non riloggarli. L'unico campo che imposti a mano è `AppVersion`, l'etichetta di release, ed è proprio ciò che più avanti rende possibile il confronto fra versioni.

Igiene e nomi dei tag

Un tag serve a filtrare, non a trasportare dati. Tienili corti, pochi e stabili nel tempo, e la dashboard resta leggibile anche dopo qualche anno di aggiunte.

In pratica:

- Usa un vocabolario piccolo e fisso: `payment`, `sync`, `ui`, `export`, `startup`. Una manciata di tag copre la maggior parte delle applicazioni. Se ti accorgi di inventare un tag nuovo ogni settimana, fermati.
- Nomina counter e id di timing in stile `feature.operation`: `report.render`, `invoice.export`,

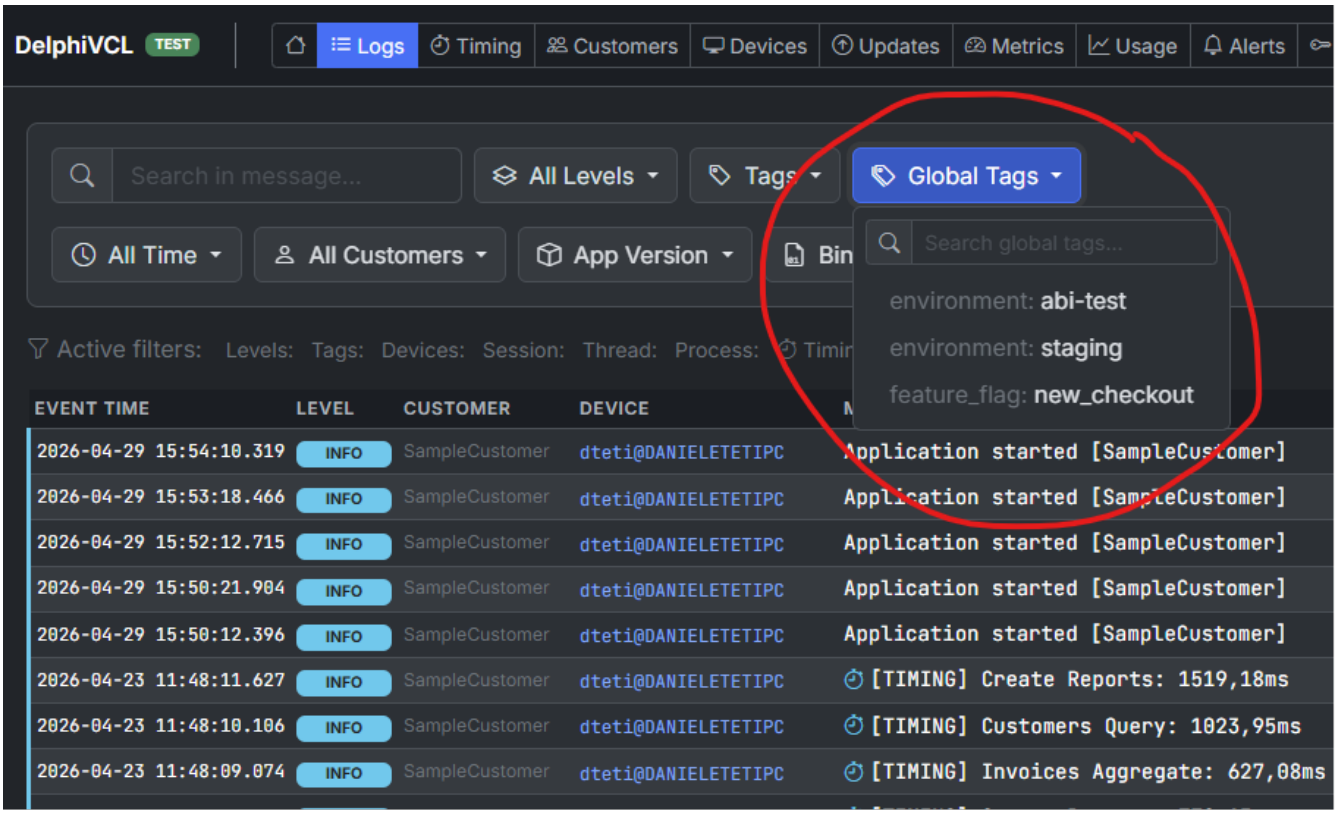
`sync.retry`. I prefissi condivisi si raggruppano da soli sulla dashboard, così tutti i tuoi numeri `sync.*` stanno insieme senza alcuna configurazione.

- Non mettere in un tag un id, un timestamp, un nome file o un valore che cambia a ogni chiamata. Ogni valore distinto diventa una dimensione a sé, quindi un tag con dentro il numero d'ordine ne crea migliaia e il filtro non serve più a niente. Si chiama alta cardinalità. Se quel valore ti serve, il posto giusto è il messaggio del log o un breadcrumb.
- Niente dati personali nei tag né nei messaggi di log, che vengono conservati ed esportati in CSV. Logga id ed esiti, non email, nomi, token o contenuti di file. Per l'identità c'è il campo id cliente.

Imposta una volta sola, all'avvio, il contesto che non cambia mai, così non lo ripeti a ogni chiamata:

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

Da quel momento ogni evento porta con sé quel contesto, e puoi filtrare l'intera dashboard fino a, per esempio, la sola edizione Pro sul canale stable senza toccare un'altra riga di codice.



I tag che imposti con `SetTag` (o con `GlobalTags` all'init) diventano un filtro "Global Tags" nella console, così puoi affettare ogni vista per ambiente, edizione, feature flag e qualunque altra cosa tu abbia taggato.

Le stesse chiamate, in ogni SDK

Le ricette che seguono usano Delphi. I concetti sono identici fra gli SDK; cambia solo la grafia. Questa tabella è la mappa, così un lettore .NET o Python può tradurre a colpo d'occhio qualunque snippet. Il riferimento nella console ha le firme complete.

Cosa vuoi	Delphi (nativo)	.NET (`EW`)	Python (`ew`)	JavaScript (`ew`)	DLL (C flat)
Loggare un errore con stack	`EW.ErrorWithException(E, 'tag')`	`EW.ErrorWithException(ex, "tag")`	`ew.error_with_exception(exc, "tag")`	`ew.error('msg', 'tag')`	`ew_ErrorWithStackTrace(…)`
Loggare a un livello	`EW.Info('msg', 'tag')`	`EW.Info("msg", "tag")`	`ew.info("msg", "tag")`	`ew.info('msg', 'tag')`	`ew_Log(level, …)`
Contare qualcosa	`EW.IncrementCounter('x', 1, 'tag')`	`EW.IncrementCounter("x", 1, "tag")`	`ew.increment_counter("x", 1, "tag")`	`ew.incrementCounter('x', 1, 'tag')`	`ew_IncrementCounter(…)`
Registrare 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 periodico	`EW.RegisterPeriodicGauge(…)`	`EW.RegisterPeriodicGauge(…)`	`ew.register_periodic_gauge(…)`	`ew.registerPeriodicGauge(…)`	*(nessuno: poll + `ew_RecordGauge`)*
Cronometrare un'operazione	`EW.StartTiming` / `EndTiming`	`EW.StartTiming` / `EndTiming`	`ew.start_timing` / `end_timing`	`ew.startTiming` / `endTiming`	`ew_StartTiming` / `ew_EndTiming`
Trace annidata	`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(…)`
Impostare un tag	`EW.SetTag('k', 'v')`	`EW.SetTag("k", "v")`	`ew.set_tag("k", "v")`	`ew.setTag('k', 'v')`	`ew_SetTag(…)`
Impostare l'id cliente	`EW.SetCustomerId('id')`	`EW.SetCustomerId("id")`	`ew.set_customer_id("id")`	`ew.setCustomerId('id')`	`ew_SetCustomerId(…)`
Impostare l'utente corrente	`EW.SetUser('id', 'email', 'name')`	`EW.SetUser("id", "email", "name")`	`ew.set_user("id", "email", "name")`	`ew.setUser({id, email})`	`ew_SetUser(…)`

Due fatti specifici degli SDK da portarsi dietro in ogni ricetta: la **DLL non ha callback per i gauge periodici**, quindi il campionamento lo piloti da un timer tuo, e **JavaScript è solo per browser**.

Tutto il resto è una semplice riscrittura uno a uno.

Ricette

Ogni ricetta parte da una situazione in cui ti sei già trovato, o ti troverai, e risale fino all'unica cosa che vale la pena instrumentare. Uno snippet Delphi canonico per ciascuna; usa la tabella qui sopra per tradurlo al tuo SDK. Dove il comportamento cambia davvero, è segnalato.

Ricetta #1: Un cliente dice che è andato in crash, e tu non hai idea del perché

> Il ticket di assistenza recita: "il programma si è chiuso da solo e ho perso il mio lavoro". Nessun passo, nessuno screenshot, nessun numero di versione. Senza telemetria la tua unica mossa è chiedere al cliente di riprodurre un crash che non riesce a riprodurre a comando, su una macchina che non puoi vedere. Metà delle volte il ticket muore lì, irrisolto, e il bug resta sul campo.

È proprio la situazione per cui esistono i log e lo stack trace, e la buona notizia è che attivarli è una riga di setup. Aggiungi `ExeWatchSDKv1.VCL` (per un'app VCL) o `ExeWatchSDKv1.FMX` (per FMX) alla clausola uses, e l'SDK aggancia per te il percorso delle eccezioni del framework: ogni eccezione GUI non gestita viene catturata come `Fatal`, taggata `exception`, con il suo stack trace, e scaricata subito così nulla va perso mentre l'app muore. Se dimentichi la unit e l'app è una GUI, l'SDK se ne accorge e ti avvisa di aggiungerla. Quando il cliente alza la cornetta, il crash è già sulla tua dashboard, con lo stack, la versione dell'app, l'OS e l'id cliente allegati. Non stai più chiedendo di riprodurre alcunché: stai leggendo cosa è successo.

Le eccezioni che catturi e gestisci tu non passano da quell'hook, quindi quelle loggale esplicitamente con l'overload a eccezione, che si porta dietro lo stack trace:

```
try
  ProcessDocument(ADoc);
except
  on E: Exception do
  begin
    EW.ErrorWithException(E, 'document');
    // gestisci o rilancia come serve alla tua app
  end;
```

```
end;
```

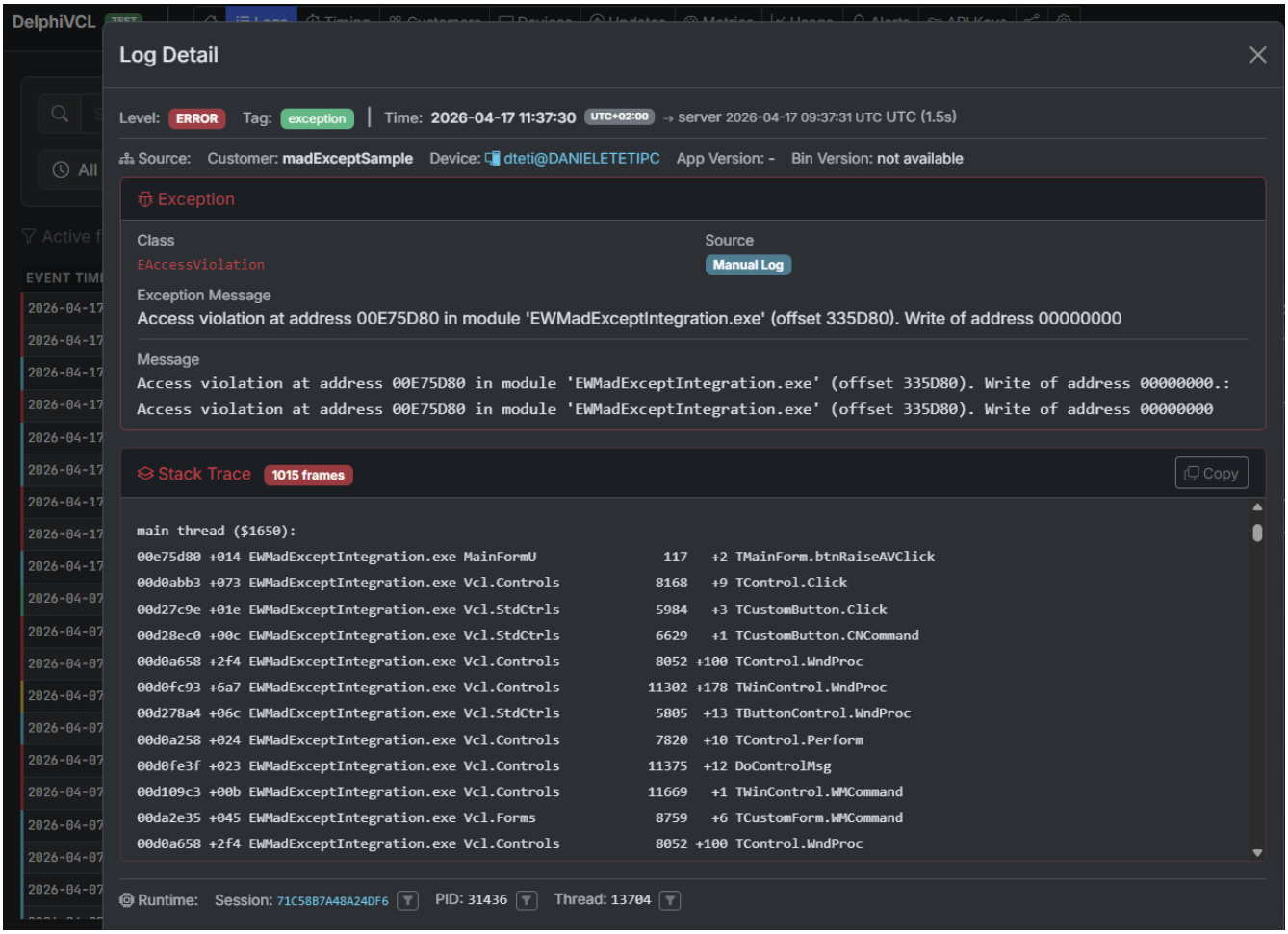
NOTE

Uno stack trace serve solo se lo sai leggere. Catturato grezzo, è un elenco di indirizzi di memoria, non nomi di metodo e numeri di riga. Per ottenere frame leggibili devi consegnare le informazioni di debug a qualunque cosa risolva lo stack. Tre modi, scegli quello adatto alla tua build:

- **Distribuisci un file MAP dettagliato.** Imposta Project Options, Linking, "Map file" su *Detailed*, e distribuisci il `.map` accanto all'eseguibile. L'SDK nativo risolve il trace confrontandolo con quello.
- **Incorpora i simboli con JclDebug.** JclDebug ripiega la map dettagliata dentro il binario stesso, così non c'è nulla di separato da distribuire. L'SDK legge le informazioni incorporate.
- **Riusa madExcept o EurekaLog se li hai già.** Risolvono lo stack in fase di link e producono un trace pienamente simbolizzato. Un ponte di una sola unit passa quel trace a ExeWatch, che mantiene lo stack fornito dal chiamante invece di sostituirlo. Vedi https://exewatch.com/ui/docs#coexisting-madexcept.

Senza nessuno di questi il crash viene comunque registrato, ma il trace resta una sequenza di indirizzi su cui non puoi agire. Fai questa cosa una volta al momento della release e ogni segnalazione di crash successiva diventa degna di essere letta.

Negli altri SDK la forma è la stessa. .NET è `EW.ErrorWithException(ex, "document")` e cattura anch'esso in automatico le eccezioni non gestite. Python è `ew.error_with_exception(…)`. La DLL espone `ew_ErrorWithStackTrace(Msg, Tag, StackTrace, ExceptionClass)`: una ABI C flat non può camminare lo stack dell'host, quindi è l'host a risolvere il trace e a passare la stringa. L'SDK JavaScript è solo browser (chiavi `ew_web_`) e cattura in automatico `window.onerror`.



Che aspetto ha un crash catturato sulla dashboard: un `EAccessViolation` con il suo stack risolto in unit, metodo e riga, accanto alla sessione, al device e al cliente che l'hanno incontrato. Questa è una build con informazioni di simbolo; senza, gli stessi frame sarebbero nudi indirizzi.

L'ultimo tassello è non dover guardare affatto la dashboard in cerca di crash. Configura un alert sul conteggio di `fatal` (trattato più avanti sotto Alert) e un picco ti raggiunge via email, in pochi minuti, invece che via ticket di assistenza, in giorni.

Ricetta #2: Hai lo stack, ma non cosa ha fatto l'utente per arrivarci

> L'errore lo conosci a memoria. Sai pure la riga: `SaveDocument`, un riferimento nil, la stessa `EAccessViolation` da tre settimane. Il punto è che capita solo da un cliente. Sulla tua macchina no, sulle altre trenta installazioni no, da lui due o tre volte a settimana. Non hai un bug da cercare, hai una sequenza da indovinare: qualcosa che quell'utente fa e gli altri no, che porta il programma in uno stato in cui quella riga esplode. Puoi passare due settimane a chiedergli "ma prima cosa stava facendo?" e

> ricevere ogni volta una ricostruzione diversa, perché nessuno ricorda davvero i propri click. Oppure puoi farlo raccontare al programma.

Lo stack trace risponde alla domanda *dove*. I breadcrumb rispondono all'altra metà, quella che ti manca sempre: **cosa stava facendo l'utente prima**. Sono briciole che lasci mentre l'applicazione lavora, ed ExeWatch le allega da sola al successivo `Error` o `Fatal`, in ordine, accanto allo stack.

Attenzione però a come si scrivono, perché è il punto in cui quasi tutti sbagliano la prima volta. Non sono cinque righe una sotto l'altra: ognuna sta dove la cosa accade davvero, sparsa nell'applicazione, e fra una e l'altra passano schermate e minuti di lavoro dell'utente.

```
// TFormFattura.FormShow -- l'utente apre una fattura
EW.AddBreadcrumb(btNavigation, 'ui', 'Aperta la fattura ' + IntToStr(AInvoiceId));

// ... qui l'utente lavora: scorre le righe, corregge un imponibile, salva ...

// TFormUtente.CambiaRuolo -- l'utente cambia ruolo
EW.AddBreadcrumb(btUser, 'auth', 'Passato al ruolo admin');

// ... qui l'utente fa altro ancora, magari per un quarto d'ora ...

// TImportListino.Esegui -- l'utente importa un file esterno
EW.AddBreadcrumb(btFile, 'io',
  Format('Importato %s (%d righe)', [ExtractFileName(AFileName), ARowCount]));

// TFatturaDAO.RicaricaRighe -- succede sotto, l'utente non lo vede nemmeno
EW.AddBreadcrumb(btQuery, 'db', 'Ricaricate le righe della fattura');

// TFormFattura.BtnEsportaClick -- l'ultimo passo prima del crash
EW.AddBreadcrumb(btClick, 'ui', 'Cliccato Esporta');
EsportaFattura(AInvoiceId);   // <-- qui salta l'eccezione, e le cinque briciole la
accompagnano
```

Nota anche i valori dentro i messaggi: il numero della fattura, il nome del file, il conteggio delle righe. Nei tag quei valori non ci vanno mai, perché ogni valore distinto diventa una dimensione e ti fa esplodere la dashboard. In un messaggio di breadcrumb invece ci stanno benissimo, ed è proprio lì che diventano utili, perché la differenza fra "importato un file" e "importato listino_2026.csv da 1284 righe" è tutta la distanza fra una traccia e una diagnosi.

Quando quel crash arriva in dashboard non stai più leggendo che c'è una access violation dentro `SaveDocument`. Stai leggendo che quel cliente, e solo quel cliente, importa un listino esterno prima di esportare, e che il tuo codice di export non ha mai visto righe con quella forma. La sequenza che non riuscivi a farti raccontare è scritta lì.

La stessa cosa serve nella versione più sfuggente del problema, quella in cui l'errore capita a molti clienti ma non sempre. Con una decina di crash in dashboard smetti di ragionare per ipotesi e cominci a confrontare le tracce: se in tutte compare un certo passaggio e nelle sessioni sane non compare mai, hai finito di indovinare. È lo stesso lavoro che faresti coi log, ma senza dover aver previsto in anticipo quale log ti sarebbe servito.

Il tipo è uno di un insieme fisso di sedici valori (`btClick`, `btNavigation`, `btHttp`, `btQuery`, `btUser`, `btForm`, `btFile`, `btState`, `btTransaction`, `btConfig`, `btCustom` e altri), e serve alla dashboard per dare un'icona e rendere la traccia leggibile a colpo d'occhio. C'è anche la forma breve, `EW.AddBreadcrumb('Export avviato')`, quando ti basta lasciare una nota.

Quattro comportamenti da conoscere, perché cambiano cosa ti ritrovi davanti quando serve:

- **La traccia è per thread, e i thread non si vedono fra loro.** Ogni thread tiene le proprie briciole. Se l'utente clicca Esporta sul thread principale e l'errore poi esplode in un thread di background, nella traccia allegata a quell'errore il click non c'è. Quando avvii un lavoro asincrono, lascia una briciola anche dentro il worker con quello che gli hai passato, altrimenti la storia si spezza esattamente nel punto in cui ti serve.
- **Restano le ultime 20 per thread.** La ventunesima fa cadere la più vecchia, quindi quello che si allega a un crash è la rincorsa immediata e non l'intera sessione. Per questo non vanno messe dentro un ciclo: venti iterazioni di `btQuery` riempiono la traccia e cancellano il contesto che le precedeva.
- **Si allegano solo a `Error` e `Fatal`.** Un `Info` o un `Debug` non se le porta dietro, e da sole non vengono mai spedite. Quindi non consumano quota finché un fallimento non le usa.
- **Vengono consumate.** Una volta allegate a un errore, la traccia di quel thread viene svuotata, così il secondo errore non si trascina la rincorsa del primo. Va però saputo: se la stessa operazione fallisce due volte di fila, il secondo evento porta solo le briciole lasciate nel frattempo. Davanti a una coppia di errori, quello con la storia completa è il primo.

Cosa vale la pena lasciare a breadcrumb: il passaggio da una schermata o da un dialog all'altro (`btNavigation`, `btForm`), le chiamate esterne che fai (`btHttp`), le query che contano (`btQuery`), le azioni con cui l'utente cambia stato al programma, login, cambio ruolo, cambio azienda (`btUser`), e i file che apre o importa (`btFile`). La regola pratica è semplice: se domani, davanti a un crash, ti verrebbe da chiedere "ma prima cosa aveva fatto?", quella è una briciola da lasciare oggi.

Cosa ti rende la dashboard: apri il crash e trovi accanto allo stack gli ultimi venti passi che ci hanno portato, in ordine, con tipo e categoria. Lo stesso crash da due clienti diversi si legge come due storie diverse, ed è lì che di solito salta fuori quale delle due è la tua.

Ricetta #3: Stai per litigare su una funzionalità che forse nessuno usa

> Una riunione di pianificazione. Qualcuno è certo che la funzionalità di export batch sia essenziale e chiede due settimane per estenderla. Qualcun altro pensa che quasi nessuno la tocchi. Entrambi tirano a indovinare, perché nessuno dei due ha un numero, e in questo genere di discussioni di solito vince la voce più alta. È esattamente il tipo di domanda che un solo counter chiude una volta per tutte.

Metti un counter al punto d'ingresso di ogni funzionalità che ti interessa. Qui lo strumento giusto è un counter, non un log, perché la domanda è "quanti", e i counter vengono sommati in rollup sul backend, così quel che rileggi è il totale su ogni installazione, a basso costo, senza che tu memorizzi una riga per clic.

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

TIP

Incrementa una volta per uso logico, non a ogni iterazione. Se generare un report elabora 500 righe, resta comunque un solo uso della funzionalità, quindi un solo incremento, non 500. Contando le righe misureresti la dimensione dei report, non quante volte la gente usa l'export, ed è la seconda la domanda a cui volevi rispondere. Se ti interessa anche la prima, quella è un gauge a parte: `EW.RecordGauge('report.rows', RowCount, 'reporting')`.

Un mese dopo la dashboard ordina le tue funzionalità per uso reale, e la discussione sulla roadmap la chiudono i fatti: estendi ciò che la gente usa e ritiri in silenzio ciò che non usa.

Ricetta #4: Una release "sembra più lenta" e nessuno riesce a dimostrarlo

> Rilasci la v4.2. Nel giro di una settimana due clienti accennano che il report principale "sembra fiacco dall'aggiornamento". È reale, o è il solito sospetto che segue ogni cambiamento? Non riesci a dirlo, perché "sembra più lenta" non è un dato, e chiedere al cliente di cronometrarlo col cronometro non è un piano.

La soluzione è misurare l'operazione sul campo e taggare ogni misura con la release che l'ha prodotta. Avvolgi l'operazione in una coppia di timing, e imposta l'etichetta di release all'init così ogni campione sa la sua versione:

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

WARNING

Chiudi sempre la coppia in un `try..finally` (o `try..except`). Se `RenderReport` solleva un'eccezione, `EndTiming` deve comunque essere eseguito, altrimenti il timing resta aperto e quel campione va perso, proprio sui percorsi d'errore che più vuoi vedere.

L'etichetta di release viene dall'init: `InitializeExeWatch(ApiKey, CustomerId, '4.2.0')` imposta `AppVersion` per l'esecuzione, mentre la versione del binario viene rilevata a parte in automatico, così ottieni entrambe. Ecco ora il flusso che risponde alla domanda: dopo che la release è stata sul campo abbastanza a lungo da raccogliere campioni, apri la pagina "Timing", filtra per finestra temporale e usa "Export CSV". Il file rispetta i filtri attivi e si apre in Excel, dove crei una pivot di durata media e p95 per `app_version`. La fiacchezza smette di essere un'opinione e diventa "report.render è passato da 120ms a 300ms nella 4.2", cosa che hai colto dai tuoi stessi dati sul campo prima che si trasformasse in abbandoni. Una vista nativa "filtra per versione" dentro l'interfaccia è nel backlog; finché non arriva, la pivot su CSV è il percorso supportato, e lo stesso export vale allo stesso modo per "Logs", "Timing" e "Metrics".

Ricetta #5: "Il checkout è lento", ma lento dove?

> I clienti dicono che il checkout ci mette una vita. Cronometri il tutto ed è quattro secondi. Quel numero da solo è quasi inutile, perché quattro secondi di cosa? Caricare il carrello? Il gateway di pagamento? Renderizzare la ricevuta? Ottimizzando alla cieca potresti passare una giornata a rendere più veloce la query di database e spostare il totale da quattro secondi a tre e nove, perché il costo vero era tutto da un'altra parte.

Le trace annidate spezzano quei quattro secondi. Avvia una trace, fai girare un timing intorno a ogni fase al suo interno, e chiudi la trace. Il backend ricostruisce una cascata, una barra per fase, così vedi quale figlio domina davvero:

```
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` restituisce un id di trace ed `EndTrace` restituisce i millisecondi totali trascorsi, così puoi loggare o fare assert sul totale se vuoi, e ogni timing annidato diventa uno span figlio. Sulla dashboard "il checkout è lento" diventa "il pagamento è l'80% del checkout", il che ti dice che il problema è il gateway, non il tuo codice, e ti risparmia la giornata che avresti passato a ottimizzare la fase sbagliata. .NET e Python rispecchiano tutto questo alla lettera; la DLL usa `ew_StartTrace(Name, Buffer, BufLen)`, che scrive l'id di trace in un buffer fornito dal chiamante, ed `ew_EndTrace(&ElapsedMs)`.

Una trace aggregata: ogni fase è una barra con il suo avg, min, max e p95, e la percentuale ti dice quanto del genitore rappresenta. Qui le fasi di transform e render dominano, quindi è lì che se ne va il tempo. Le statistiche sono calcolate sulle esecuzioni riuscite, così una fase fallita non le distorce.

Cosa succede se dimentichi di chiudere un timing

Prima o poi un `EndTiming` lo dimenticherai. L'SDK è costruito per sopravvivergli senza perdere memoria né riportare un numero sbagliato, ma quel che ottieni non è mai buono quanto una coppia pulita, ed è la vera ragione del try/finally. Ecco esattamente cosa capita a un timing aperto, a

seconda di come va a finire:

- **Niente, se resta semplicemente aperto.** Un timing senza un `EndTiming` corrispondente non emette mai un campione. Non è contato né mediato, semplicemente assente. Perdi in silenzio quella singola misura.
- **Auto-chiuso come fallito se riavvii lo stesso id.** Chiama `StartTiming('report.render')` mentre un `report.render` precedente è ancora aperto sullo stesso thread e l'SDK chiude per te il vecchio, marcato `success = false` e segnalato `auto_closed`, con un Warning nei tuoi log ("auto-closed, duplicate StartTiming"). Essendo fallito, resta fuori dalle statistiche di durata; il Warning c'è per dirti che il codice ha un buco.
- **Il più vecchio viene sfrattato se se ne accumulano troppi.** Ogni thread tiene al massimo 100 timing aperti. Avviane un 101esimo e quello aperto più vecchio viene chiuso forzatamente come fallito, di nuovo con un Warning. È la rete di sicurezza che impedisce a una lenta fuga di timing dimenticati di crescere senza limite.
- **Una trace ripulisce i propri figli.** Se il timing dimenticato è annidato dentro uno `StartTrace` /`EndTrace`, `EndTrace` chiude forzatamente qualunque span figlio tu abbia lasciato aperto, marcato fallito, così la cascata resta comunque completa.

Lo schema in tutti e quattro i casi: un `EndTiming` dimenticato diventa un campione fallito più un Warning, mai una durata pulita. È il meccanismo che ti protegge, non una funzionalità su cui appoggiarti. Avvolgi ogni coppia in un try/finally e niente di tutto questo scatta mai.

Ricetta #6: L'app va bene alle 9 e arranca alle 17

> Un cliente segnala che la tua applicazione è scattante al mattino e fiacca a fine giornata, e un riavvio la sistema fino all'indomani. Quel comportamento è una fuga di risorse, ed è invisibile a un crash reporter, perché non va in crash niente. Degrada e basta, in silenzio, nell'arco di ore, su una macchina che non vedi mai.

Una fuga è un livello che sale e non torna più giù, che è esattamente ciò che misura un gauge. Non vuoi spargere chiamate di gauge per tutto il tuo loop di render; vuoi il valore campionato a intervalli. Registra un gauge periodico e il thread di campionamento dell'SDK legge la tua callback per te, senza alcun timer tuo:

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

```
  end,
  'runtime');
```

Cose buone da osservare così: working set in MB, numero di GDI o di handle, numero di documenti aperti, dimensione della cache. Dopo un giorno o due la dashboard mostra la firma con chiarezza, un dente di sega che sale per tutta la sessione e crolla al riavvio, e poiché un gauge tiene media, min, max e numero di campioni per finestra, la media e il max in crescita sono la fuga. Affetta per `app_version` e spesso riesci a inchiodarla alla release esatta che l'ha introdotta.

Una divergenza da rispettare: la callback del gauge periodico esiste su Delphi, .NET (un `Func<double>`), Python e JS, ma non sulla DLL, perché una callback non può attraversare la ABI C flat. Con la DLL l'intervallo lo possiede l'host: fai girare un timer tuo e chiama `ew_RecordGauge(Name, Value, Tag)` a ogni tick.

Ricetta #7: È tutti, o un solo cliente?

> I tuoi log mostrano una raffica di errori di sync e il primo istinto è che tutto il parco macchine stia fallendo. Prima di farti prendere dal panico, chiediti se sta colpendo tutti o un solo cliente con una configurazione insolita, e nota che scorrere log grezzi da 200 installazioni non te lo dirà mai.

Se imposti l'id cliente all'init e taggi i log per sottosistema, puoi isolare pulitamente qualsiasi fetta:

```
EW.SetCustomerId('acme-corp');
// ... più avanti, nel percorso di sincronizzazione:
EW.Error('Sincronizzazione rifiutata dal server', 'sync');
```

Ora filtra la dashboard sugli errori `sync` di Acme e il quadro si chiarisce in pochi secondi: è un solo cliente, dietro un firewall aziendale, non il tuo codice che fallisce dappertutto. Fai un passo in più e crea un alert con ambito `customer_external_id` e tag, e ti arriva la notifica specifica sui problemi di sync di Acme, restando in silenzio sugli altri 200 clienti che stanno bene. Tassi d'errore per cliente e per funzionalità, ciascuno con un proprio alert, tutti frutto di una taggatura coerente.

Ricetta #8: Con che frequenza fallisce davvero?

> "Il sync a volte fallisce" è il genere di segnalazione che nasconde l'unica cosa che ti serve sapere: quanto spesso è "a volte"? Una volta a settimana è un'alzata di spalle; un tentativo su tre è un incidente. Non puoi dargli una

> priorità finché non vedi il rapporto.

La versione più semplice è un counter per esito, affettato con un tag di esito:

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

Ma se stai già cronometrando l'operazione, non ti serve un secondo counter, perché `EndTiming` porta un flag `success` come terzo parametro (default `True`):

```
EW.StartTiming('sync', 'sync');
try
  DoSync;
  EW.EndTiming('sync', nil, True);   // riuscito
except
  EW.EndTiming('sync', nil, False);  // fallito, ma il timing si chiude lo stesso
  raise;
end;
```

Questo flag fa qualcosa di più sottile di un counter, e vale la pena capirlo. Un timing fallito non viene scartato: viene registrato, così puoi contare i fallimenti e vederli nella lista dei timing. Ma viene lasciato fuori dalle statistiche di durata. Media, min, max e p95 nella pagina "Timing" sono calcolati solo dalle esecuzioni riuscite. Conta, perché i fallimenti sono spesso le chiamate più lente, un'operazione che ha rinunciato dopo un timeout di 30 secondi, e se quelle contassero nella tua latenza penseresti che il sync sia rallentato quando in realtà ha cominciato a fallire. Con il flag success leggi la vera latenza delle esecuzioni riuscite e il tasso di fallimento, da una sola chiamata.

Counter, gauge o timing: quale

Questi tre strumenti vengono confusi di continuo, e confonderli non lancia un errore, registra in silenzio dati che non significano niente, il che è peggio. La distinzione è semplice, una volta detta ad alta voce:

Strumento	Risponde a	Come aggrega	Cosa ti costa sbagliare
Counter	quanti, quanto spesso	sommato sulla finestra	un gauge per "numero di export" butta via il totale

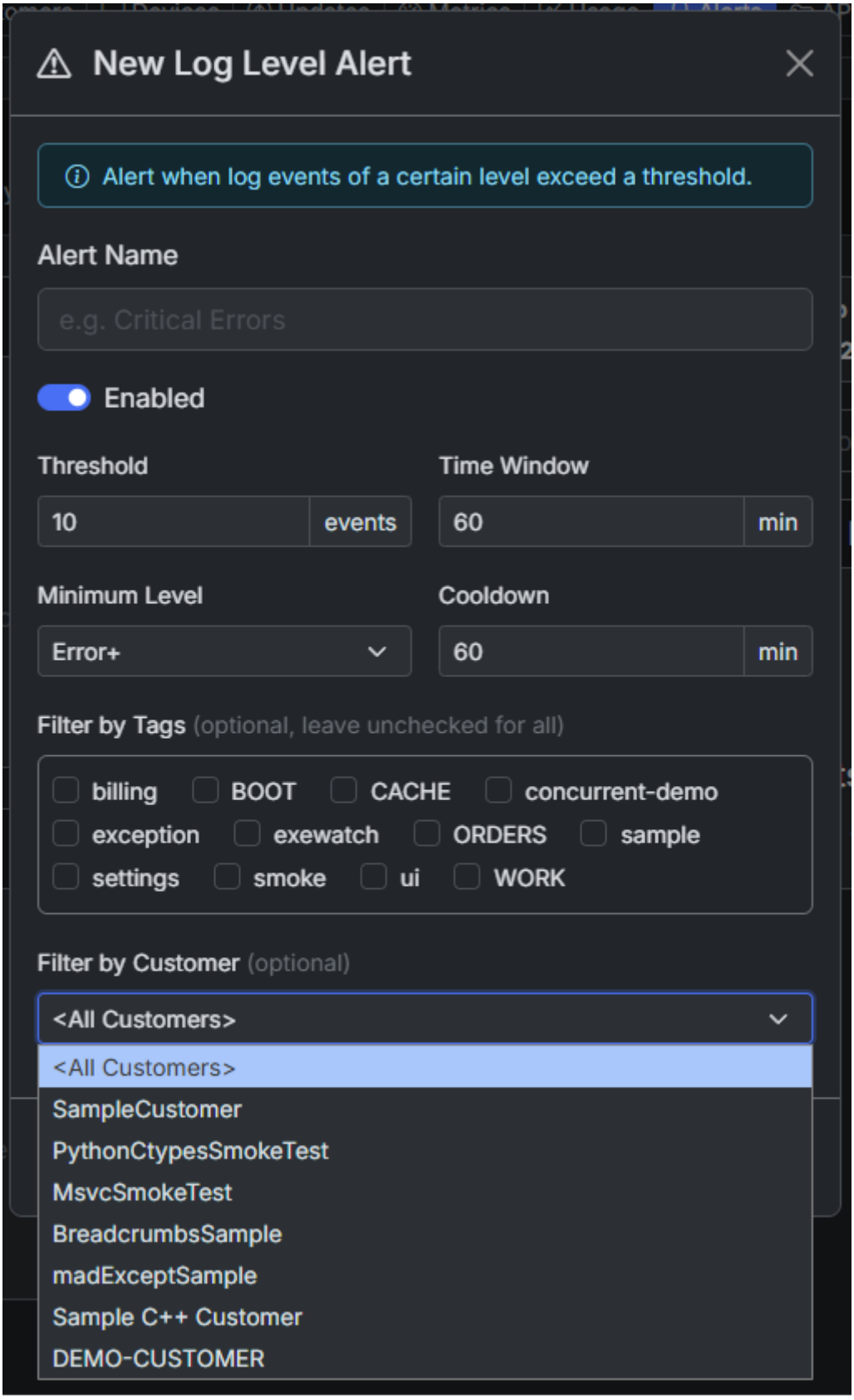
Strumento	Risponde a	Come aggrega	Cosa ti costa sbagliare
Gauge	qual è il valore in questo momento	media, min, max, numero di campioni per finestra	un counter per "profondità di coda" somma livelli in un'assurdità
Timing	quanto ci ha messo	distribuzione di durata, con trace	un counter per "quanto lento" ti dà un conteggio, non una durata

Nel dubbio, prova a sommare due letture. Se la somma ha senso, 12 export più 8 export fanno 20 export, è un counter. Se non ha senso, una coda di 12 più una coda di 8 non fa una coda di 20, è un gauge. Se quello che ti interessa è il tempo trascorso, è un timing.

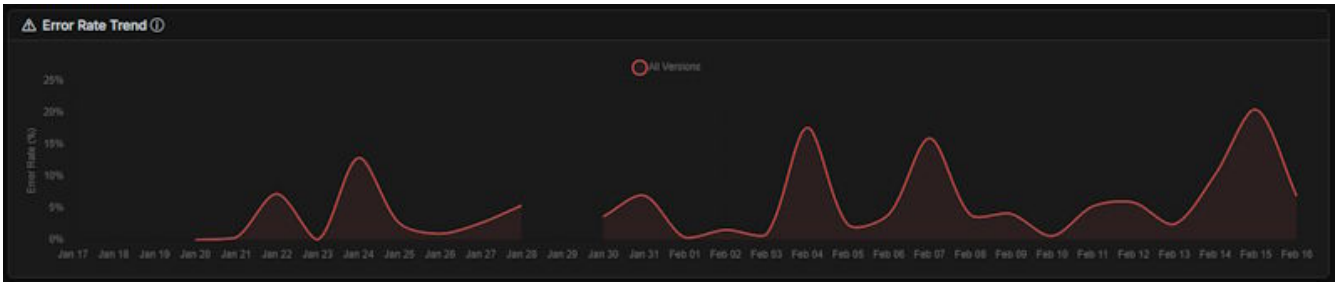
Alert che contano

Un alert è ciò che ti permette di smettere di guardare la dashboard. Invece di controllare ogni tanto sperando di cogliere un problema, descrivi il problema una volta ed ExeWatch ti manda una email quando accade. Oggi ne esistono due tipi, e conviene sapere con precisione su cosa scatta ciascuno, perché i limiti onesti dell'alerting plasmano il modo in cui usi tutto quel che viene sopra.

Gli alert sul volume di log scattano quando il conteggio degli eventi di log pari o superiori a un livello supera una soglia dentro una finestra. Imposti `threshold` (un conteggio), `window_minutes`, `min_level` (default `error`), un filtro `tags` opzionale, un `customer_external_id` opzionale e `cooldown_minutes` (default 60) così un solo incidente non ti notifica cinquanta volte. Uno tipico recita: "più di 20 eventi `fatal` in 15 minuti per il cliente Acme".



Lo stesso alert come form nella console, in "Configure alerts": una soglia su una finestra temporale a un livello minimo, con un cooldown, all'occorrenza ristretto per tag e per cliente. Impostare "Filter by Customer" su Acme è il modo per farti avvisare degli errori di Acme senza sentir parlare di nessun altro.



Tasso d'errore nel tempo. Un picco come quelli qui è esattamente ciò che un alert sul volume di log sorveglia, così ti raggiunge via email nell'istante in cui parte invece che quando ti capita di aprire questo grafico.

Gli alert di timing scattano quando le operazioni che corrispondono a un pattern di id di timing girano più lente di una soglia di durata, abbastanza spesso da superare una soglia di conteggio dentro una finestra. I campi sono un `timing_id_pattern`, una soglia di durata in millisecondi, una soglia di occorrenze `threshold` (default 5), `window_minutes`, un `customer_external_id` opzionale e `cooldown_minutes` (default 240). È così che ti fai avvisare che "report.render sta girando lento sul campo" senza doverlo sorvegliare.

NOTE

È altrettanto importante sapere cosa gli alert oggi non fanno. Non esiste un "alert quando un gauge supera X" né un "alert quando un counter supera X". Gli alert scattano su conteggi di eventi di log e sul timing, non su valori arbitrari di metrica. Quindi se vuoi essere notificato quando la memoria o la profondità di coda supera una soglia, la risposta onesta è che non puoi ancora; guardi il gauge sulla dashboard. L'alerting a soglia su valori di metrica al momento non è una funzionalità.

Il modo per tenere gli alert utili invece che ignorati: mantieni onesti i tuoi livelli di log (vedi anti-pattern, altrimenti la tua soglia `error` scatta su cose che errori non sono), dai a ogni alert un ambito per tag o per cliente così ha un titolare chiaro, e imposta un cooldown abbastanza lungo perché un vero incidente arrivi come una sola email.

Confrontare le versioni con l'export CSV

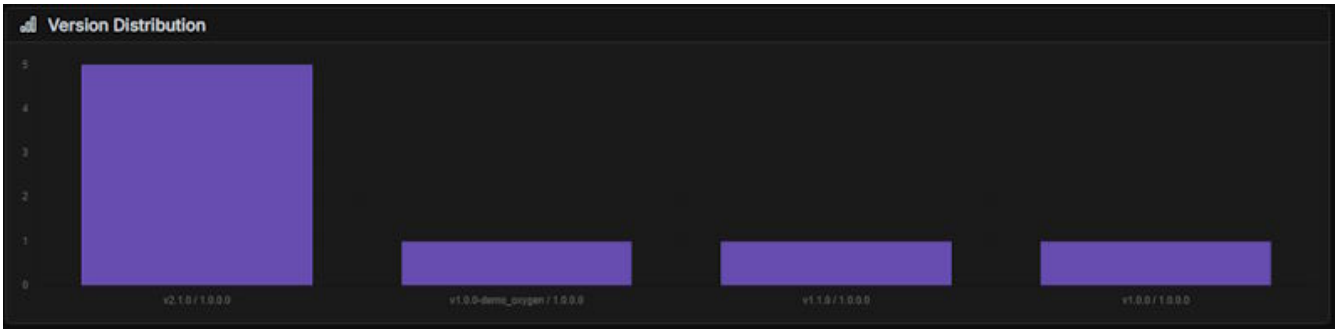
Rilasci una versione e vuoi sapere se è diventata più lenta o più rumorosa della precedente. Oggi quel confronto passa dall'export CSV, e vale la pena esplicitare il flusso perché risponde a una domanda che i clienti pongono di continuo.

1. Rilascia la versione con `AppVersion` impostato all'init (`InitializeExeWatch(ApiKey, CustomerId, '4.2.0')`). Ogni evento ora sa quale versione l'ha prodotto.
2. Dagli tempo sul campo per raccogliere campioni, poi apri la pagina "Timing" e imposta i filtri:

finestra temporale, tag, cliente.

3. Usa "Export CSV". Il file rispetta quei filtri attivi e si apre in Excel.
4. Crea una pivot di durata media e p95 per `app_version`, e la regressione, se c'è, è lì nella tabella pivot.

Lo stesso export c'è nelle pagine "Logs" e "Metrics", così puoi confrontare il volume d'errori o i livelli di metrica fra le versioni allo stesso modo. Un filtro di timing per-versione dentro l'interfaccia è nel backlog; quando arriverà, questo diventerà un paio di clic invece di una pivot. Fino ad allora, la strada del CSV è il modo supportato per il confronto versione su versione, e funziona abbastanza bene che parecchi team si fermano qui.



Poiché ogni evento porta il suo `AppVersion`, la console sa già come sono distribuite le tue release sul campo. Quella stessa etichetta di versione è ciò su cui crei la pivot del timing esportato.

Mettiamo tutto insieme: instrumentare un metodo reale

Le chiamate isolate sono facili da approvare con un cenno e difficili da collocare. Ecco quindi l'instrumentazione calata in codice ordinario: un metodo di sync su un data module, il tipo che ogni app Delphi gestionale ha. Prima la versione nuda, che fa il suo lavoro senza telemetria:

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

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

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

Ora lo stesso metodo instrumentato, ogni aggiunta fa esattamente un lavoro:

```
procedure TSyncModule.SyncInvoices;
var
  Response: IHTTPResponse;
begin
  EW.AddBreadcrumb(btHttp, 'sync', 'GET /invoices since ' + FLastSync);  // traccia
per un eventuale crash
  EW.StartTiming('sync.invoices', 'sync');                               // cronometra
l'intera operazione
  try
    Response := FHttp.Get(FBaseUrl + '/invoices?since=' + FLastSync);
    if Response.StatusCode <> 200 then
      raise ESyncError.CreateFmt('Sincronizzazione rifiutata dal server: HTTP %d',
[Response.StatusCode]);

    EW.StartTiming('sync.import', 'sync');                               // isola la
fase di database
    FConnection.StartTransaction;
    try
      ImportInvoices(Response.ContentAsString);
      FConnection.Commit;
      EW.EndTiming('sync.import', nil, True);
    except
      FConnection.Rollback;
      EW.EndTiming('sync.import', nil, False);                           // esecuzione
fallita, fuori dalle statistiche
      raise;
    end;

    FLastSync := NowUtcIso;
    EW.IncrementCounter('sync.completed', 1, 'sync');                    // uno per
ogni sync riuscita
    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;
```

Cosa è andato dove, e perché:

- Il **breadcrumb** sta in cima, prima della chiamata che descrive, così che se qualcosa più a valle solleva un'eccezione, la traccia registri già la richiesta che vi ha condotto.
- Il **timing esterno** (`sync.invoices`) avvolge l'intera operazione; il **timing interno** (`sync.import`) isola la fase di database, così la cascata separa il tempo di rete dal tempo di import.
- Il **server che risponde male solleva un'eccezione**, come qualsiasi altro guasto: con uno stato diverso da 200 non c'è niente da importare, e chi ha chiamato deve sapere che la sincronizzazione non è avvenuta. Nota che questo non aggiunge instrumentazione, la toglie: il fallimento HTTP finisce nello stesso gestore in fondo, che lo logga con la sua classe d'eccezione e i breadcrumb, invece di avere un ramo che si scrive addosso log e chiusure di timing.
- Ogni `EndTiming` sta sia sul percorso di successo sia su quello di fallimento, con `False` quando l'operazione fallisce, così un sync rotto non inquina i tuoi numeri di latenza.
- Il **counter** incrementa una volta, solo in caso di successo, così `sync.completed` è un conteggio veritiero dei sync riusciti.
- L'**errore** viene loggato con l'eccezione nel gestore più esterno, dove porta lo stack e il breadcrumb qui sopra, poi rilanciato così la gestione errori dell'app resta invariata.

Nota la forma: l'instrumentazione incornicia il codice reale, non lo rimpiazza. Cancella ogni riga `EW.` e il metodo funziona ancora esattamente come prima. La instrumentazione si limita a osservare.

Anti-pattern

Nessuno di questi è "hai loggato troppo". Sono i casi in cui quello che mandi ti fuorvia invece di aiutarti, e la correzione sta accanto a ciascuno.

Ripetere la stessa riga dentro un ciclo. Una chiamata `Debug` in un ciclo caldo, o un `IncrementCounter` a ogni iterazione. Il punto non è la quantità: è che cinquecento righe identiche rispondono tutte alla stessa domanda a cui aveva già risposto la prima, e nel frattempo consumano la quota che serviva agli eventi veri. Registra l'operazione, non le sue iterazioni: una riga quando parte, una quando finisce o fallisce, e un solo incremento di counter per operazione logica. Se il dettaglio per-iterazione ti serve davvero durante una diagnosi, tienilo a livello `Debug` e alza il livello minimo dell'applicazione dalla console quando hai finito.

Dati personali nei log. Email, nomi, token o contenuti di file in un messaggio o in un tag. I log

vengono conservati ed esportati in CSV, e i tag sono dimensioni a bassa cardinalità, non payload, quindi questo è insieme un problema di privacy e un pasticcio. Logga id e categorie; usa il campo id cliente per l'identità.

Livelli che mentono. Tutto loggato a `Error`, oppure fallimenti veri loggati a `Info`. Gli alert usano di default `min_level = error`, quindi se i tuoi livelli sono sbagliati lo sono anche i tuoi alert, e o vieni notificato per niente o non vieni mai notificato. La scala che li tiene onesti: Fatal significa che l'app non può continuare, Error che un'operazione è fallita, Warning degradato ma funzionante, Info un cambio di stato notevole, Debug dettaglio per soli sviluppatori.

Gauge e counter, scambiati. Un counter per "profondità di coda attuale" somma livelli in un totale privo di senso; un gauge per "numero di export" butta via il totale. Applica il test dell'addizione della tabella qui sopra prima di scegliere.

Tag che esplodono. Un id, un timestamp, o un qualsiasi valore per-richiesta in un tag. I tag sono dimensioni con un piccolo insieme fisso di valori; valori illimitati si moltiplicano in migliaia di dimensioni usa-e-getta e rendono la dashboard inutilizzabile. Tieni piccolo il vocabolario.

Riloggare ciò che ottieni già gratis. Loggare a mano OS, versione o hardware che già viaggiano nello snapshot del device a ogni batch. Non farlo. L'unico campo che imposti a mano è `AppVersion`, l'etichetta di release.

Cosa succede se cade la connessione a internet?

Le app desktop girano su portatili che si infilano nei tunnel, su macchine dietro VPN capricciose, su un PC che qualcuno stacca dalla rete alla chiusura. Quindi la domanda è legittima: che ne è della tua telemetria quando l'SDK non riesce a raggiungere il server?

Non si perde niente. L'SDK non spedisce gli eventi direttamente dal punto in cui li chiami. Li accumula in un buffer, li scrive su disco, e a spedirli ci pensa un thread shipper in background. Quando la rete è giù l'invio semplicemente fallisce, i file restano su disco, e lo shipper riprova a intervalli. Nell'istante in cui la connettività torna, i file in coda vengono spediti in ordine. Un crash loggato in aereo compare sulla tua dashboard la prima volta che il portatile torna online.

A governarlo sono quattro impostazioni di `TExeWatchConfig`:

- `StoragePath` è dove vivono i file in attesa. Di default è una cartella per-app; puntalo in un posto in cui la tua app possa sempre scrivere.
- `FlushIntervalMs` (default 5000) è ogni quanto il buffer in memoria viene scritto su disco. Più

corto significa meno dati a rischio se il processo viene ucciso a metà corsa; più lungo significa scritture meno frequenti e più grandi.

- `RetryIntervalMs` (default 30000) è ogni quanto lo shipper riprova dopo un fallimento. È la tua cadenza di riconnessione: un valore più basso smaltisce prima l'arretrato una volta tornata la rete, al costo di più tentativi mentre è ancora giù.
- `MaxPendingAgeDays` (default 7) limita per quanto tempo i file non spediti vengono tenuti. Una macchina offline più a lungo di così scarta i dati più vecchi invece di spedire alla riconnessione un'ondata vecchia di settimane. Mettilo a 0 per tenere tutto, senza limite.

L'arretrato puoi tenerlo d'occhio tu stesso: `GetPendingCount` restituisce quanti eventi sono ancora in attesa di spedizione.

NOTE

Una connessione caduta non è un `429`, anche se entrambi finiscono col dato che non arriva al server. Sono opposti. Un guasto di rete è temporaneo, quindi l'SDK tiene il dato e riprova. Un `429` vuol dire che hai superato la quota, il che è voluto, quindi l'SDK scarta il dato invece di ritentare. Perdere telemetria per una connessione ballerina richiede un blackout più lungo di `MaxPendingAgeDays`; perderla per un `429` richiede solo di sforare il tuo piano, ed è di questo che parla la prossima sezione.

Instrumenta ciò che ha significato, al livello giusto

Un fatto tecnico da conoscere, e arriva alla fine della guida di proposito: la quota mensile conta gli eventi che ci mandi, non quelli che restano archiviati. Contano insieme i log e gli aggiornamenti delle metriche, cioè counter, gauge e timing, non i soli log. I messaggi interni dell'SDK, quelli con tag `ew.system`, non rientrano nel conteggio. E siccome conta quello che è arrivato, cancellare dati libera spazio ma non restituisce quota.

Non è un invito a instrumentare di meno. L'ordine giusto è quello che hai seguito fin qui: prima decidi cosa vuoi vedere, poi, se e quando il volume diventa un tema, lo governi. Gli strumenti per governarlo esistono e nessuno ti chiede di rinunciare a un dato che ti interessa.

- Il livello minimo si imposta per applicazione dalla console, e l'SDK lo riceve alla connessione successiva. Puoi sviluppare con tutto il `Debug` che vuoi e poi tenere in produzione solo `Info` e oltre, senza toccare il codice e senza rilasciare una nuova build.
- Sempre dalla console, il campionamento manda solo una frazione degli eventi di routine. `Error` e `Fatal` lo scavalcano sempre, quindi puoi diradare un log verboso su diecimila installazioni senza perdere un solo crash.

- Aggrega ciò che è ripetizione pura: un incremento per operazione logica invece di uno per iterazione di ciclo.
- Usa i tag per affettare, così una metrica taggata bene ti risparmia dieci metriche quasi identiche.

Se sei stabilmente vicino al tetto con un'instrumentazione che ti serve tutta, non c'è niente da tagliare: stai seguendo più applicazioni o più clienti di quando hai scelto il piano, e a quel punto conviene passare al livello successivo.

> **NOTE** Superata la quota mensile l'SDK riceve un `429` e comincia a scartare i dati, quindi conviene tenere d'occhio l'utilizzo nella console. Non per loggare meno, ma per accorgerti in tempo che il piano ti sta stretto, invece di scoprirlo perdendo gli eventi di un picco.

Sulla roadmap

Alcune cose che la gente chiede sono pianificate ma non ancora rilasciate. Sono elencate qui, onestamente, così sai dove sono oggi i confini e non vai a cercare una funzionalità che ancora non c'è.

- **Rilevare quando un'app diventa silenziosa.** Un dead-man's-switch, così vieni a sapere del cliente la cui app ha smesso di farsi viva, non solo di quello la cui app ha sollevato un errore. Oggi gli alert scattano su eventi che accadono, non su eventi che smettono di accadere, quindi questo non è ancora possibile. Pianificato.
- **Rilevamento di anomalie.** Far emergere in automatico un "qui c'è qualcosa di strano" sulle metriche, invece che impostare tu soglie fisse. Non ancora realizzato.
- **Una pagina di usage analytics.** Una vista dedicata all'adozione e ai pattern d'uso delle funzionalità, oltre i counter grezzi. Non ancora realizzata.
- **Una API di lettura / query.** Oggi le API key sono solo di ingest, quindi tirare fuori i tuoi dati per reportistica automatizzata aspetta questa. L'export CSV è la strada manuale nel frattempo.

Per tutto ciò che oggi esiste, il riferimento dell'SDK nella console ha le firme esatte, le chiavi di configurazione e i passi d'installazione. Questa guida è il cosa e il perché; quello è il come.