Stop guessing. Start knowing.

Der ExeWatch DevGuide

Version 1.50.1

Ein ExeWatch-SDK in eine Delphi-Anwendung einzubauen kostet ein paar Zeilen: Der Schnellstart weiter unten ist schon alles, was du brauchst. Sobald die Integration steht und das Dashboard lebendig wird, kommt die schwierigere Frage, die, die das Referenzhandbuch nicht beantwortet: Was lohnt es sich wirklich im Auge zu behalten?

Genau diese Frage beantwortet dieser Leitfaden. Er steht eine Ebene über der technischen Referenz (die "Docs"-Seite in der Konsole, unter https://exewatch.com/ui/docs). Die Referenz erklärt, wie du das SDK installierst, initialisierst und aufrufst; hier geht es darum, was sich zu rufen lohnt, warum und wann. Der rote Faden sind Situationen, die du sofort wiedererkennst: der Support-Anruf wegen eines Absturzes, den niemand reproduzieren kann, das Feature, von dem du nicht weißt, ob es überhaupt jemand nutzt, das Release, das sich "langsamer anfühlt", ohne dass es jemand beweisen kann.

Schnellstart

Eine `uses`-Klausel und eine Zeile Initialisierung. Der ganze Rest des Leitfadens ist Feinschliff an diesen beiden Dingen.

1. Füge das SDK deiner uses-Klausel hinzu. In einer VCL-Anwendung, also der großen Mehrheit der Delphi-Apps, sind es zwei Units: Die zweite fängt die Exceptions der GUI ab.

```
uses
  ExeWatchSDKv1, ExeWatchSDKv1.VCL;   // FireMonkey-App: ExeWatchSDKv1.FMX statt .VCL
```

Von den beiden Hook-Units nimmst du genau eine, die des Frameworks, das du verwendest. Hat die App keine GUI, etwa ein Windows-Dienst oder ein Kommandozeilenwerkzeug, dann füge weder `.VCL` noch `.FMX` hinzu: Es gibt keine Message Loop zum Einhängen, und `ExeWatchSDKv1` allein reicht dir.

2. Initialisiere einmal, früh (im `.dpr` vor `Application.Run` oder im `OnCreate`-Ereignis des Hauptformulars):

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

Zwei Argumente: der API-Key, den du aus der Konsole kopierst, und die ID des Kunden, zu dem diese Installation gehört. Fertig, die App wird überwacht.

Was dir diese zwei Zeilen gebracht haben. Ab jetzt werden Abstürze mit ihrem Stack Trace erfasst, ohne dass du weiteres schreibst, und die beiden Units teilen sich die Arbeit. `ExeWatchSDKv1` hängt sich in `System.ExceptProc` ein, wo die unbehandelten Exceptions des normalen Codes landen. `ExeWatchSDKv1.VCL` (oder `.FMX`) nimmt dagegen die, die innerhalb eines Ereignisses entwischen, eines `OnClick` oder eines `OnCreate`: Die fängt die Message Loop des Frameworks ab und übergibt sie an `Application.OnException`, und dort endet der Weg, bei `System.ExceptProc` kommen sie nie an. In einer Desktop-Anwendung sind sie die häufigste Absturzkategorie, und deshalb zählt die zweite Unit.

Außerdem geht ein automatischer Log `Application started` raus, die Bestätigung, dass die Integration lebt, und die Geräteinformationen (Betriebssystem, Maschine, Version des Binaries) werden gesendet und dem Kunden zugeordnet, wo sie in der Konsole unter "Devices" auftauchen. Nichts davon läuft auf dem Thread deiner Anwendung: Die Events landen in einer Warteschlange auf der Platte, und ein Hintergrund-Thread verschickt sie, ein langsames Netz oder ein nicht erreichbarer Server bremst oder blockiert die App also nicht.

Die Hook-Unit nimmt dir deinen bisherigen Handler nicht weg: Hattest du bereits ein eigenes `Application.OnException`, loggt der Hook und ruft es danach auf; hattest du keines, ruft er `Application.ShowException`. Das Fehlerfenster, das dein Benutzer vorher sah, erscheint unverändert weiter. Und wenn du sie irgendwann in einem anderen Projekt vergisst, stehst du nicht im Dunkeln: Das SDK merkt, dass es in einer GUI-Anwendung ohne installierten Hook läuft, und schreibt selbst ein `Warning` mit Tag `exewatch`, das du im Dashboard wiederfindest.

IMPORTANT

Der Stack Trace kommt immer, die Zeilennummern nicht: Die brauchen die `.map`-Datei. Der Delphi-Compiler lässt Namen von Units, Methoden und Zeilen nicht in der ausführbaren Datei; er schreibt sie separat, in eine `.map`-Datei neben dem Binary. Das SDK sucht sie beim Start unter dem Namen der ausführbaren Datei (`MyApp.exe` → `MyApp.map`) und macht damit aus Adressen lesbare Frames. Findet es sie nicht, wird der Absturz trotzdem erfasst, aber der Trace ist eine Spalte nackter Adressen wie `[00007FF6A21C3F40]`.

Um sie zu bekommen, einmalig: Setze Project Options, Building, Delphi Compiler, Linking, "Map file" auf *Detailed* (die anderen Stufen, *Segments* und *Publics*, enthalten keine Zeilennummern; mit *Publics* bekommst du die Methodennamen, aber nicht die Zeilen). Liefere die `.map` dann zusammen mit der ausführbaren Datei aus, im selben Ordner.

Du kannst sie nicht mitliefern? Das ist eine legitime Entscheidung: Die `.map` ist eine Textdatei von einigen Megabyte, die die internen Namen deines Codes offenlegt. Archiviere in dem Fall die `.map` jedes Release zusammen mit dem Binary, das du ausgeliefert hast, denn die Adressen, die du im

> Dashboard siehst, bleiben später auflösbar, aber nur mit der Map genau dieses Builds: Neu kompilieren verschiebt sie, und die neue Map nützt nichts. Wer keine getrennten Dateien verwalten will, hat zwei weitere Wege: JclDebug, das die Symbole in die ausführbare Datei selbst faltet, und madExcept oder EurekaLog, falls du sie ohnehin nutzt, die den Stack zur Link-Zeit auflösen und ExeWatch einen bereits symbolisierten Trace übergeben. Beide findest du weiter unten, in der Notiz zu Rezept #1.

3. Logge, was dich interessiert. Ab hier loggst, zählst und misst du die Zeit, wann immer du etwas Aufzeichnenswertes hast:

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

Öffne das Dashboard und die Events sind da. Von hier an hilft dir der Leitfaden dabei, auf alles zu kommen, was dort hineingehört, denn die Liste ist länger als die, die dir jetzt spontan einfallen würde.

Dann, eine Einstellung nach der anderen

Nichts von dem, was folgt, brauchst du für den Start: Füge es hinzu, wenn du es wirklich brauchst, eine Zeile nach der anderen.

Du willst wissen, aus welchem Release ein Event kommt? Übergib ein drittes Argument.

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

Das ist `AppVersion`, dein Release-Label ('4.2.0', '2026-Q1', 'v2-beta'). Die Version des Binaries ist ein eigenes Feld, das das SDK schon selbst aus der ausführbaren Datei liest: Dieses dritte Argument brauchst du, wenn deine Vorstellung von Release nicht mit der Nummer übereinstimmt, die in der `.exe` kompiliert ist.

Beim Start weißt du noch nicht, wer der Kunde ist? Initialisiere trotzdem, mit leerer ID, und setze sie, sobald du sie kennst. Wie und warum steht im Abschnitt *Initialisierung: über den Einzeiler hinaus*, ein Stück weiter unten.

Jedes SDK integrieren

Der Schnellstart war Delphi, das Flaggschiff-SDK. Das *Was* und *Warum* im Rest des Leitfadens gilt

für jedes SDK gleich: Nur das Einbinden und Initialisieren unterscheidet sich. Hier ist die Einrichtung für jedes. In allen ist das letzte Argument das Release-Label (`AppVersion`), und der API-Key trägt das Präfix der Plattform (`ew_win_`, `ew_lin_`, `ew_web_` und so weiter).

Delphi (nativ). Wie im Schnellstart: `ExeWatchSDKv1` in der uses-Klausel, dazu `ExeWatchSDKv1.VCL` (oder `.FMX`), wenn die App eine GUI hat, dann `InitializeExeWatch(ApiKey, CustomerId, '4.2.0')`. Die VCL/FMX-Unit erfasst unbehandelte GUI-Exceptions für dich.

.NET. Referenziere das `ExeWatch`-Paket (füge `ExeWatch.WinForms` für eine WinForms-Anwendung hinzu), dann initialisiere einmal beim Start:

```
using ExeWatch;

EW.Initialize("ew_win_xxxxxxxx", "acme-corp", "4.2.0");
ExeWatchWinForms.Install();   // nur WinForms, vor Application.Run()

EW.Info("Anwendung gestartet", "startup");
```

Eine Konsolen- oder Service-Anwendung lässt die `Install`-Zeile weg: Der Client hängt sich selbst an `AppDomain.UnhandledException`. WinForms braucht dagegen den expliziten `Install`-Aufruf vor `Application.Run`.

Python. Installiere das SDK, initialisiere das Singleton und nutze danach das modulweite `ew`:

```
from exewatch import initialize_exewatch, ew

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

ew.info("Dienst gestartet", "startup")
ew.increment_counter("job.run", 1, "jobs")
```

Python hat keine GUI zum Einhängen, also erfasse Fehler dort, wo du sie behandelst, mit `ew.error_with_exception(exc, "tag")`, oder richte `sys.excepthook` auf das SDK aus, um ein globales Netz zu spannen.

JavaScript (Browser). Du deklarierst die Konfiguration in `window.ewConfig` und lädst dann das Skript von exewatch.com. Der Key ist ein `ew_web_`-Key, und das SDK erfasst `window.onerror` automatisch:

```
<!-- zuerst die Konfiguration... -->
<script>
  window.ewConfig = {
    apiKey: 'ew_web_xxxxxxxx',
    customerId: 'acme-corp',
```

```
    appVersion: '4.2.0'
  };
</script>

<!-- ...dann das Skript, ausgeliefert von exewatch.com -->
<!-- Produktion (minifiziert, 12 KB) -->
<script src="https://exewatch.com/static/js/exewatch.v1.min.js"></script>

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

Die Reihenfolge zählt: Das SDK initialisiert sich beim `DOMContentLoaded` selbst und liest dabei `window.ewConfig`, die Konfiguration muss also schon auf der Seite stehen, wenn das Skript geladen wird. Das Skript lieferst du von exewatch.com aus, nicht aus einer eigenen Kopie, so erreichen Korrekturen deine Benutzer, ohne dass du etwas neu verteilen musst.

Dieses SDK ist nur für den Browser, einen Node-Build gibt es nicht. Nicht abgefangene Fehler werden für dich erfasst, zusammen mit fehlgeschlagenen `fetch`- und `XHR`-Aufrufen und den `console.error`-Ausgaben. Auf einer Seite, die Skripte von Dritten lädt, verwirft `ignoreUrls` in derselben `ewConfig` die Fehler aus Domains, die du nicht kontrollierst, etwa Werbung und Analytics: Das ist Rauschen, das dir nichts über deine Anwendung sagt.

DLL (C, C++, VB, altes .NET, alles mit einem C-ABI). Lade `ExeWatchSDKv1DLL.dll` und rufe die flachen Exports auf. Strings sind Wide (`PWideChar`), jede Funktion ist `stdcall`, und Ergebnisse kommen als Rückgabecodes zurück:

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

Weil ein flaches C-ABI weder den Stack des Hosts durchlaufen noch einen Callback ausführen kann, fallen dem Host zwei Aufgaben zu: einen bereits aufgelösten Stack-String an `ew_ErrorWithStackTrace` übergeben und periodische Gauges über deinen eigenen Timer ansteuern (es gibt keinen Callback für periodische Gauges). Ein Delphi-Host, der die DLL den nativen Units vorzieht, kann die Import-Unit `ExeWatchSDKv1Imports` verwenden und `EWInitialize(…)` statt `InitializeExeWatch` aufrufen.

Dasselbe, vollständig, mit MSVC. Keine Import-Library und kein Embarcadero-Runtime: Definiere `EW_DYNAMIC_LOAD` vor dem Header, und jedes `ew_*` wird zu einem Funktionszeiger, den `ExeWatchSDKv1.dynload.c` zur Laufzeit mit `LoadLibrary` und `GetProcAddress` auflöst. Das hier ist ein vollständiges Programm, kein Ausschnitt:

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

```
int wmain()
{
    // sucht ExeWatchSDKv1DLL_x64.dll im Standard-Suchpfad von Windows
    if (ew_LoadSDK() != EW_OK)
    {
        fwprintf(stderr, L"ExeWatchSDKv1DLL_x64.dll nicht gefunden\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 fehlgeschlagen: %ls\n", err);
        ew_UnloadSDK();
        return 1;
    }

    ew_Info(L"Beispielanwendung gestartet", L"startup");
    ew_IncrementCounter(L"report.generated", 1.0, L"reporting");

    ew_WaitForSending(15);   // liefert die Restmenge in der Queue: 0 = alles
verschickt
    ew_Shutdown();
    ew_UnloadSDK();
    return 0;
}
```

Kompiliert wird mit einem einzigen Befehl, aus einer "x64 Native Tools Command Prompt for VS 2022", unter Angabe des Ordners mit Header und Loader:

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

Die einzige Zeile, die hier keine Zeremonie ist, ist `ew_WaitForSending`. Ein Kommandozeilenwerkzeug kann enden, bevor der Shipper-Thread die Queue geleert hat, und dieser Aufruf schreibt den Puffer auf die Platte und wartet, bis die Queue leer ist, wobei er zurückgibt, wie viele Events noch warten. Ein Datenverlust ist das nicht, denn die Queue liegt auf der Platte und läuft beim nächsten Lauf weiter, aber ohne das Warten erscheinen deine Events erst eine Runde später im Dashboard. Das kompilierbare Sample mit allem Übrigen, Benutzeridentität, globale Tags, Breadcrumbs, verschachtelte Timings und Gauges, liegt in ExeWatchSamples/MSVCWithDLLSDK.

Dieselbe DLL aus einer Delphi-Konsole. Auch ein Delphi-Host kann statt der nativen Units die DLL verwenden, und in zwei Fällen ist das die richtige Wahl: wenn du auf einem Delphi älter als XE8 arbeitest, dem Minimum des nativen SDK, und wenn du mehrere Anwendungen aktualisieren musst, indem du ein einziges Binary verteilst. Die Import-Unit `ExeWatchSDKv1Imports` reicht zurück

bis Delphi 5. Auch das hier ist ein ganzes Programm:

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

  EWInfo('Nachtlauf gestartet', 'batch');
  EWIncrementCounter('report.generated', 1.0, 'reporting');

  LRemaining := ew_WaitForSending(15);
  if LRemaining > 0 then
    WriteLn('Es bleiben ', LRemaining, ' Events in der Queue: Sie gehen beim nächsten
Lauf raus.');

  ew_Shutdown;
end.
```

Die Wrapper mit dem Präfix `EW` nehmen `string` entgegen und erledigen die Umwandlung nach `PWideChar` selbst, in deinem Code steht also kein Cast. Weil es eine Konsole ist, gelten dieselben zwei Zeilen wie vorhin: `ew_WaitForSending` vor dem Beenden und `ew_Shutdown` zum sauberen Schließen.

Eine Sache solltest du vor dem Ausliefern wissen. Standardmäßig bindet die Unit die DLL statisch, `ExeWatchSDKv1DLL_x64.dll` muss beim Start also neben der ausführbaren Datei liegen: Fehlt sie, startet der Prozess überhaupt nicht, Windows bricht mit einem Fehler wegen fehlender DLL ab, noch vor deiner ersten Codezeile. Soll die Anwendung stattdessen trotzdem starten und nur auf die Telemetrie verzichten, definiere `EW_DYNAMIC_LOAD` in den Projektoptionen: Die Unit wechselt zu `LoadLibrary`, und du rufst beim Start `EWLoadDLL` auf und prüfst dessen Ergebnis, so wie es das MSVC-Beispiel weiter oben macht.

Und mit Free Pascal. Die DLL hat nichts Delphi-Spezifisches: ein flaches C-ABI, `stdcall`, Wide Strings. Unter Windows kompiliert dieselbe `ExeWatchSDKv1Imports` mit FPC, den die Unit erkennt und selbst in `{$MODE DELPHI}` versetzt. Wo `string` nicht Unicode ist, wird der interne Alias zu `WideString`, und die `EW*`-Wrapper konvertieren für dich, der Code, den du schreibst, bleibt also der aus dem Beispiel oben.

Initialisierung: über den Einzeiler hinaus

Der einzelne `InitializeExeWatch`-Aufruf aus dem Schnellstart ist alles, was die meisten Anwendungen je brauchen. Das dritte Argument ist `AppVersion`, dein eigenes Release-Label ('4.2.0', '2026-Q1', 'v2-beta'); die Binärversion ist ein separates, automatisch aus der ausführbaren Datei erkanntes Feld, du bekommst also beides aus der einen Zeile. Hier ist, wonach du greifst, wenn diese Zeile nicht ausreicht.

TIP

Du kennst den Kunden noch nicht? Initialisiere trotzdem. In den meisten echten Anwendungen sollte das SDK ab der ersten Zeile live sein, damit ein Absturz beim Start erfasst wird, aber zu welchem Kunden diese Installation gehört, erfährst du erst, nachdem du eine Lizenzdatei gelesen hast oder der Benutzer sich angemeldet hat. Initialisiere mit einer leeren Customer-ID und setze sie, sobald du sie kennst.

```
// im .dpr, vor Application.Run, damit das SDK sofort live ist
InitializeExeWatch('ew_win_xxxxxxxx', '', '4.2.0');

// ... später, wenn du den Kunden kennst (aus einer Lizenzdatei oder nach dem Login):
EW.SetCustomerId(Session.CustomerCode);
```

Das ist ein bewusster, unterstützter Ablauf, kein Workaround. Solange die Customer-ID leer ist, hält das SDK den Device-Info-Datensatz zurück, weil es die ID braucht, um das Gerät dem richtigen Kunden zuzuordnen, und sendet ihn in dem Moment, in dem du `SetCustomerId` aufrufst. Wenn sich die ID später ändert (ein anderer Kunde auf derselben Maschine), sendet das SDK die Device-Info erneut unter dem neuen Kunden, sodass das Gerät für beide korrekt auftaucht. Die Regel ist einfach: erst initialisieren, dann identifizieren, sobald du kannst.

Kunde und Benutzer sind zwei verschiedene Identitäten, gesetzt mit zwei verschiedenen Aufrufen. `SetCustomerId` setzt den *Kunden*: das Konto oder den Mandanten, zu dem diese Installation gehört, also das Kriterium, nach dem das Dashboard Geräte und Alerts gruppiert. Wer die Anwendung tatsächlich benutzt, ist eine andere Sache, der *Benutzer*, und den setzt du mit `SetUser`:

```
// nachdem sich die Person angemeldet hat:
EW.SetUser('u-8842', 'mario.rossi@acme.example', 'Mario Rossi');
// beim Abmelden:
EW.ClearUser;
```

ID, E-Mail und Name reisen dann mit jedem Event als `user_id` mit, sodass ein Absturz zeigt, welcher

Kunde ihn getroffen hat und welche Person. Verwende `SetCustomerId` für das Konto und `SetUser` für den Menschen: Sie sind unabhängig, und du kannst das eine, das andere, beide oder keines setzen.

WARNING

> `SetCustomerId` **und** `SetUser` **gelten beide für den ganzen Prozess: Sie passen zu einer Single-User-Anwendung, nicht zu einem Multi-User-Server.** Jeder ist ein einzelner Wert auf der SDK-Instanz für den ganzen Prozess, nicht etwas, das an einen Thread oder einen Request gebunden ist. Das ist genau richtig für eine Desktop-Anwendung, in der jeweils ein Kunde und ein angemeldeter Benutzer aktiv sind. Falsch ist es für eine serverseitige Anwendung, die viele Benutzer gleichzeitig bedient: Zwei Requests auf zwei Threads würden die Identität des jeweils anderen überschreiben, und Events würden demjenigen zugeschrieben, der sie zuletzt gesetzt hat. ExeWatch ist für Desktop- und Single-User-Anwendungen gebaut. Auf einem mandantenfähigen Server verfolge die Identität pro Request nicht auf diese Weise; trage sie im Event selbst mit (ein Tag pro Aufruf oder ein `extra_data`-Feld).

Wenn der Drei-Argument-Aufruf nicht ausreicht, baue eine `TExeWatchConfig` und übergib die stattdessen. Jedes Feld hat einen sinnvollen Standardwert, setze also nur, was du brauchst:

```
var
  Config: TExeWatchConfig;
begin
  Config := TExeWatchConfig.Create('ew_win_xxxxxxxx', 'acme-corp');
  Config.AppVersion := '4.2.0';
  Config.SampleRate := 0.25;                // sendet 25% der Routine-Events;
Error/Fatal immer
  Config.GaugeSamplingIntervalSec := 60;    // wie oft periodische Gauges gelesen
werden (Standard 30, min 10)
  Config.MaxPendingAgeDays := 3;            // verwirft Dateien in der Queue, die
älter sind (Standard 7)
  Config.AnonymizeDeviceId := True;         // ersetzt den Benutzernamen in der
Device-ID durch einen Hash (DSGVO / AD)
  Config.GlobalTags := [TPair<string, string>.Create('edition', 'pro')];
  InitializeExeWatch(Config);
end;
```

Die Einstellungen, die man kennen sollte:

- **SampleRate** (0 bis 1): der Anteil der Routine-Events, die tatsächlich gesendet werden. `Error` und `Fatal` umgehen das Sampling immer, du kannst also Info- und Debug-Volumen auf einer großen Flotte ausdünnen, ohne je einen Absturz zu verlieren.
- **GlobalTags** und **InitialCustomDeviceInfo**: Tags und Gerätefelder, die vor dem allerersten Event angewendet werden, sodass sogar der automatische "Application started"-Log bereits

deinen Kontext trägt.

- **GaugeSamplingIntervalSec**: wie oft der Sampler-Thread deine periodischen Gauges liest (Standard 30s, Untergrenze 10s).
- **MaxPendingAgeDays**: während die App offline ist, sammeln sich Events auf der Platte an; Dateien, die älter als dieser Wert sind, werden gelöscht, sodass eine wochenlang offline gebliebene Maschine beim erneuten Verbinden keine veralteten Daten sendet (Standard 7).
- **AnonymizeDeviceId**: ersetzt den Benutzernamen-Teil der Device-ID durch einen Hash, für DSGVO- oder Active-Directory-Umgebungen.
- **Endpoint**: richtet das SDK auf eine selbst gehostete ExeWatch-Instanz statt auf die Standard-Cloud aus. Nur On-Premise.

Die anderen SDKs übernehmen dieselben Konzepte unter sehr ähnlichen Feldnamen; die Referenz hat die genaue Signatur für jedes.

Warum es diesen Leitfaden gibt

Ein leeres Dashboard ist einschüchternd, und die erste Frage ist immer dieselbe: Wo fange ich an? Die kurze Antwort: Wenn dich etwas interessiert, gehört es aufgezeichnet. Willst du wissen, ob diese Funktion fehlschlägt? Logge. Wie oft sie benutzt wird? Zähle. Ob sie langsam ist und wie sehr? Miss die Zeit. Der teure Fehler ist nicht, ein paar Events zu viel geschickt zu haben. Der teure Fehler ist, vor einem Problem in Produktion zu stehen und festzustellen, dass ausgerechnet dieses Stück Code nichts erzählt.

Dieser Moment kommt immer, und er kommt an einem Dienstagnachmittag mit einem Kunden am Telefon. Du hast den Stack Trace des Absturzes, aber du weißt nicht, was der Benutzer einen Augenblick vorher getan hat, weil du diesen Schritt nicht geloggt hattest. Du weißt, dass die Synchronisation ewig dauert, aber nicht, welche Phase sie frisst, weil du nur das Gesamt gemessen hattest. In diesen zwanzig Minuten bereust du nie die Zeilen, die du zu viel geschickt hast. Du bereust die drei, die du nicht geschrieben hast.

Fang also großzügig an. Wenn etwas fehlschlagen kann, wenn es langsam werden kann, wenn du wissen musst, wie oft es benutzt wird, wenn es erklärt, warum die App sich so verhalten hat, dann schick es. Solange du den Code schreibst, ist Hinzufügen leicht; später hinzufügen, wenn der fragliche Build schon bei dreihundert Kunden installiert ist, heißt ein Release und Wochen Wartezeit. Und das Rauschen, falls es irgendwann zu viel wird, nimmst du jederzeit weg: Mindest-Level und Sampling jeder Anwendung änderst du in der Konsole, ohne etwas neu zu kompilieren.

Das Einzige, was sich nicht zu schicken lohnt, ist das, was nicht einmal dir etwas sagt: ein

`Debug('bin hier')`, ein `Info('step 1')`, eine Nachricht, die dir beim Schreiben nur deshalb nützlich vorkommt, weil du den Kontext in diesem Moment im Kopf hast. Wenn du sie Monate später in einer Log-Liste wiederfindest, ist dieser Kontext weg, und die Zeile hat jeden Zweck verloren. Der Rest des Leitfadens hilft dir, darauf zu kommen, was stattdessen hineingehört, und für jede Sache das passende Werkzeug zu wählen, denn ein Absturz, eine Nutzungszählung und eine Dauer werden auf drei verschiedene Arten erfasst.

Die fünf Fragen

Vor den Rezepten die Landkarte. ExeWatch gibt dir fünf Werkzeuge, und jedes existiert, um eine andere Frage zu beantworten. Fast jede Entscheidung der Art "Was erfasse ich hier?" wird offensichtlich, sobald du weißt, welche Frage du stellst.

Die Frage, die du hast	**Das Werkzeug, das sie beantwortet**	**Worauf du es richtest**
Was ist passiert?	Logs (debug bis fatal, mit Tag und Stack Trace)	einzelne Ereignisse, die ein Mensch lesen möchte: Fehler, Zustandsänderungen, "der Benutzer tat X"
Wie viele, wie oft?	Counter	Dinge, die du zählst und aufsummierst: Feature-Nutzungen, Retries, Exporte, Fehlschläge
Was ist der Wert gerade jetzt?	Gauges	ein Pegel, der steigt und fällt: Speicher in MB, Queue-Tiefe, offene Dokumente, Cache-Größe
Wie lange hat es gedauert?	Timing (und verschachtelte Traces)	Dauern von Operationen, mit Traces, um eine langsame in ihre Phasen zu zerlegen
Sag mir, wenn etwas nicht stimmt	Alerts	ein Schwellwert auf Error- oder Fatal-Log-Volumen oder auf Operationsdauer, in der Konsole gesetzt, nicht im Code

Zwei Dinge übersieht man leicht, und beide sparen dir später echte Arbeit.

Tags sind das stille sechste Werkzeug. Jeder Log-, Counter-, Gauge- und Timing-Aufruf nimmt einen `tag`, und es gibt ein prozessweites `SetTag` für Kontext, der mit allem mitläuft. Tags sind, wie du das Dashboard in "payment" gegen "sync" gegen "ui" zerschneidest, ohne für jedes Feature eine eigene Metrik zu erfinden. Mach sie richtig und jedes Diagramm lässt sich so filtern, wie du denkst; mach sie falsch und das Dashboard verkommt zu Rauschen. Der Abschnitt zur Tag-Hygiene kommt als Nächstes, und er ist kurz.

Device-Info ist automatisch und kostenlos. App-Version, Binärversion, Betriebssystem und Hardware werden mit jedem Batch mitgeschickt, ohne einen einzigen Aufruf von dir. Logge sie nicht erneut. Das eine Feld, das du von Hand setzt, ist `AppVersion`, das Release-Label, und genau das macht später Versionsvergleiche möglich.

Tag-Hygiene und Benennung

Ein Tag dient zum Filtern, nicht zum Transportieren von Daten. Halte sie kurz, wenige und über die Zeit stabil, dann bleibt das Dashboard auch nach ein paar Jahren voller Ergänzungen lesbar.

In der Praxis:

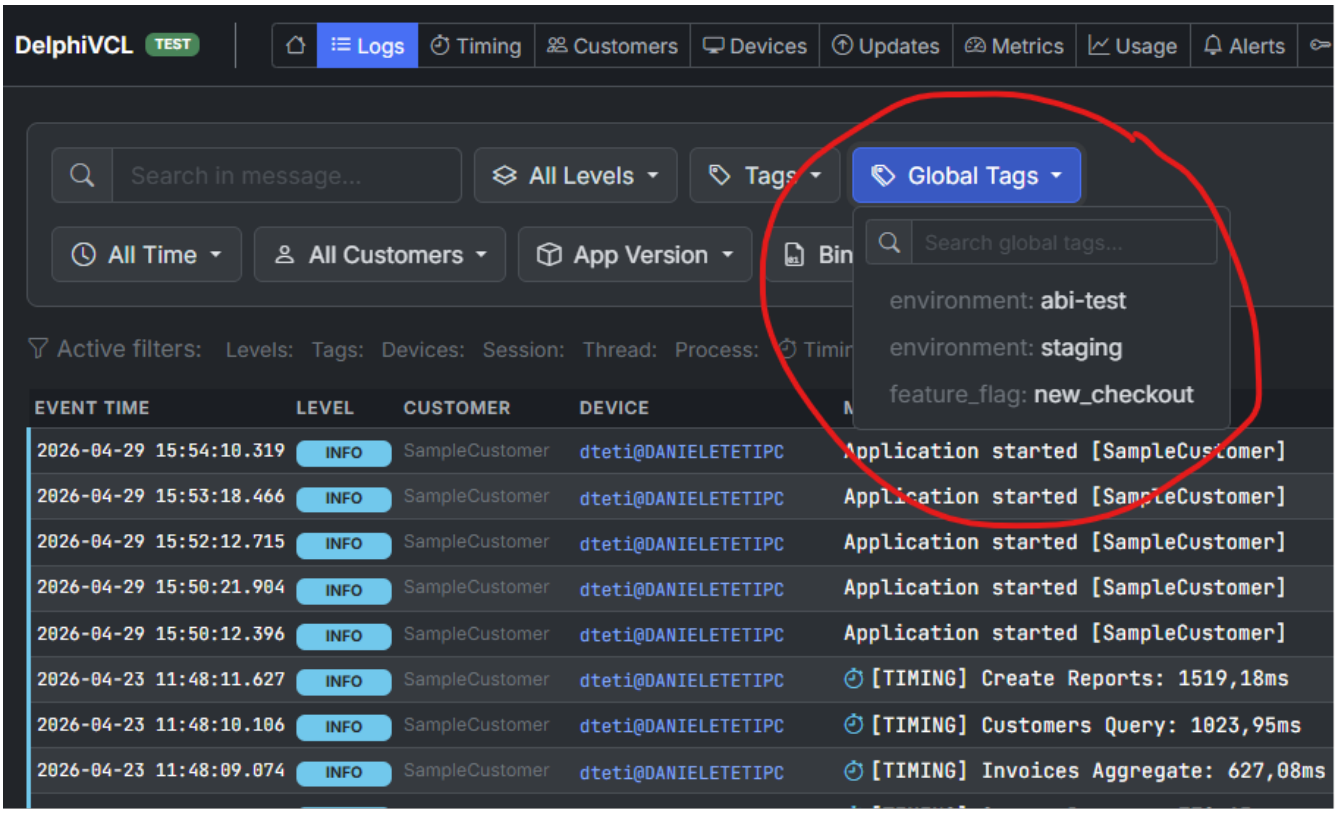
- Nutze ein kleines, festes Vokabular: `payment`, `sync`, `ui`, `export`, `startup`. Eine Handvoll Tags deckt die meisten Anwendungen ab. Wenn du dich dabei ertappst, jede Woche ein neues Tag zu erfinden, hör auf.
- Benenne Counter- und Timing-IDs im Stil `feature.operation`: `report.render`, `invoice.export`, `sync.retry`. Die gemeinsamen Präfixe gruppieren sich von selbst im Dashboard, sodass all deine `sync.*`-Zahlen ohne jede Konfiguration zusammenstehen.
- Steck keine ID, keinen Zeitstempel, keinen Dateinamen und keinen Wert, der sich bei jedem Aufruf ändert, in ein Tag. Jeder unterschiedliche Wert wird zu einer eigenen Dimension, ein Tag mit der Bestellnummer darin erzeugt also Tausende davon, und der Filter nützt nichts mehr. Das heißt hohe Kardinalität. Wenn du diesen Wert brauchst, gehört er in die Log-Nachricht oder in einen Breadcrumb.
- Keine personenbezogenen Daten in Tags oder Log-Nachrichten, die aufbewahrt und nach CSV exportiert werden. Logge IDs und Ergebnisse, nicht E-Mails, Namen, Tokens oder Dateiinhalte. Für die Identität gibt es das Customer-ID-Feld.

Setze den Kontext, der sich nie ändert, einmal beim Start, damit du ihn nicht bei jedem Aufruf wiederholst:

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

```
EW.SetTag('channel', 'stable');
```

Von da an trägt jedes Event diesen Kontext, und du kannst das ganze Dashboard filtern auf, sagen wir, die Pro-Edition auf dem Stable-Channel, ohne eine weitere Codezeile anzurühren.



Tags, die du mit `SetTag` (oder `GlobalTags` bei der Initialisierung) setzt, werden in der Konsole zu einem "Global Tags"-Filter, sodass du jede Ansicht nach Umgebung, Edition, Feature-Flag und allem anderen, was du getaggt hast, aufschlüsseln kannst.

Dieselben Aufrufe, in jedem SDK

Die folgenden Rezepte nutzen Delphi. Die Konzepte sind über alle SDKs hinweg identisch; nur die Schreibweise ändert sich. Diese Tabelle ist die Landkarte, sodass ein .NET- oder Python-Leser jeden Ausschnitt auf einen Blick übersetzen kann. Die Konsolen-Referenz hat die vollständigen Signaturen.

Was du willst	Delphi (nativ)	.NET (`EW`)	Python (`ew`)	JavaScript (`ew`)	DLL (flaches C)
Fehler mit Stack loggen	`EW.ErrorWithException(E, 'tag')`	`EW.ErrorWithException(ex, "tag")`	`ew.error_with_exception(exc, "tag")`	`ew.error('msg', 'tag')`	`ew_ErrorWithStackTrace(…)`

Was du willst	Delphi (nativ)	.NET (EW)	Python (ew)	JavaScript (ew)	DLL (flaches C)
Auf einem Level loggen	`EW.Info('msg', 'tag')`	`EW.Info("msg", "tag")`	`ew.info("msg", "tag")`	`ew.info('msg', 'tag')`	`ew_Log(level, …)`
Etwas zählen	`EW.IncrementCounter('x', 1, 'tag')`	`EW.IncrementCounter("x", 1, "tag")`	`ew.increment_counter("x", 1, "tag")`	`ew.incrementCounter('x', 1, 'tag')`	`ew_IncrementCounter(…)`
Einen Gauge erfassen	`EW.RecordGauge('x', v, 'tag')`	`EW.RecordGauge("x", v, "tag")`	`ew.record_gauge("x", v, "tag")`	`ew.recordGauge('x', v, 'tag')`	`ew_RecordGauge(…)`
Periodischer Gauge	`EW.RegisterPeriodicGauge(…)`	`EW.RegisterPeriodicGauge(…)`	`ew.register_periodic_gauge(…)`	`ew.registerPeriodicGauge(…)`	*(keiner: pollen + `ew_RecordGauge`)*
Eine Operation messen	`EW.StartTiming` / `EndTiming`	`EW.StartTiming` / `EndTiming`	`ew.start_timing` / `end_timing`	`ew.startTiming` / `endTiming`	`ew_StartTiming` / `ew_EndTiming`
Verschachtelter Trace	`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(…)`
Ein Tag setzen	`EW.SetTag('k', 'v')`	`EW.SetTag("k", "v")`	`ew.set_tag("k", "v")`	`ew.setTag('k', 'v')`	`ew_SetTag(…)`
Die Customer-ID setzen	`EW.SetCustomerId('id')`	`EW.SetCustomerId("id")`	`ew.set_customer_id("id")`	`ew.setCustomerId('id')`	`ew_SetCustomerId(…)`
Den aktuellen Benutzer setzen	`EW.SetUser('id', 'email', 'name')`	`EW.SetUser("id", "email", "name")`	`ew.set_user("id", "email", "name")`	`ew.setUser({id, email})`	`ew_SetUser(…)`

Zwei SDK-spezifische Fakten, die man in jedes Rezept mitnehmen sollte: Die **DLL hat keinen Callback für periodische Gauges**, also steuerst du das Sampling über deinen eigenen Timer an, und **JavaScript läuft nur im Browser**. Alles andere ist ein Eins-zu-eins-Umschreiben.

Rezepte

Jedes Rezept startet aus einer Situation, in der du warst oder noch sein wirst, und arbeitet zurück zu der einen Sache, die es zu instrumentieren lohnt. Jeweils ein kanonischer Delphi-Ausschnitt; nutze die Tabelle oben, um ihn in dein SDK zu übersetzen. Wo sich das Verhalten wirklich unterscheidet, ist es vermerkt.

Rezept #1: Ein Kunde sagt, es sei abgestürzt, und du hast keine Ahnung warum

> Das Support-Ticket lautet: "Das Programm hat sich selbst geschlossen und ich habe meine Arbeit verloren." Keine Schritte, kein Screenshot, keine Versionsnummer. Ohne Telemetrie ist dein einziger Zug, den Kunden zu bitten, einen Absturz zu reproduzieren, den er nicht auf Kommando reproduzieren kann, auf einer Maschine, die du nicht sehen kannst. Die Hälfte der Zeit stirbt das Ticket dort, ungelöst, und der Bug bleibt im Feld.

Das ist genau die Situation, für die Logs und ein Stack Trace existieren, und die gute Nachricht ist, dass das Einschalten eine Zeile Einrichtung ist. Füge `ExeWatchSDKv1.VCL` (für eine VCL-App) oder `ExeWatchSDKv1.FMX` (für FMX) deiner uses-Klausel hinzu, und das SDK hängt sich in den Framework-Exception-Pfad für dich ein: Jede unbehandelte GUI-Exception wird als `Fatal` erfasst, mit `exception` getaggt, samt Stack Trace, und sofort geflusht, sodass nichts verloren geht, während die App stirbt. Wenn du die Unit vergisst und die App eine GUI ist, bemerkt das SDK das und warnt dich, sie hinzuzufügen. Bis der Kunde zum Hörer greift, liegt der Absturz bereits auf deinem Dashboard, mit dem Stack, der App-Version, dem Betriebssystem und der angehängten Customer-ID. Du bittest niemanden mehr, irgendetwas zu reproduzieren; du liest, was passiert ist.

Die Exceptions, die du selbst fängst und behandelst, durchlaufen diesen Hook nicht, also logge sie explizit mit dem Exception-Overload, der den Stack Trace mitträgt:

```
try
  ProcessDocument(ADoc);
except
  on E: Exception do
  begin
    EW.ErrorWithException(E, 'document');
    // behandle oder löse erneut aus, wie es deine App braucht
  end;
end;
```

NOTE

Ein Stack Trace ist nur nützlich, wenn du ihn lesen kannst. Roh erfasst ist er eine Liste von Speicheradressen, keine Methodennamen und Zeilennummern. Um lesbare Frames zu bekommen, musst du die Debug-Info an das übergeben, was den Stack auflöst. Drei Wege, wähle den, der zu deinem Build passt:

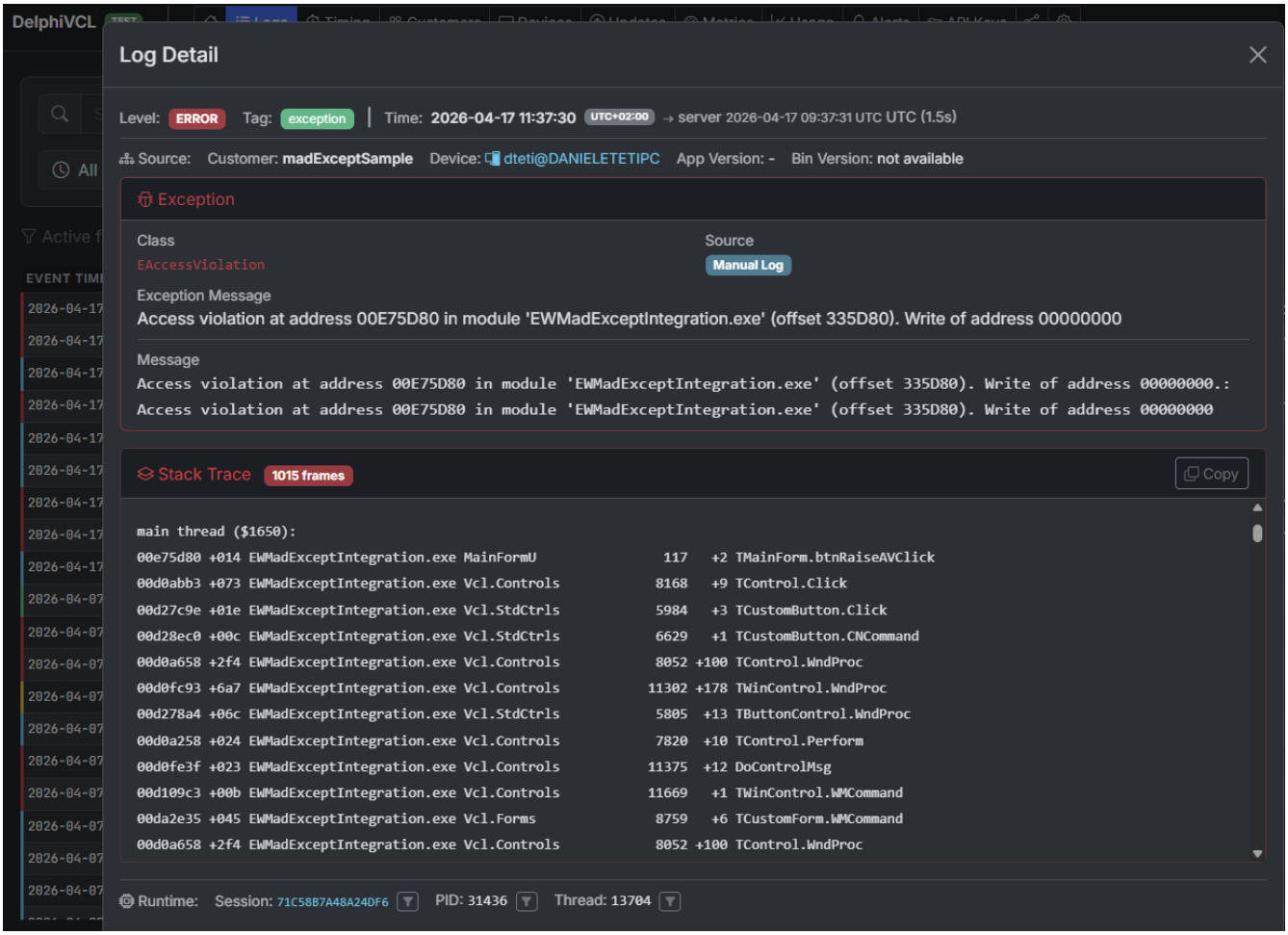
- **Liefere eine detaillierte MAP-Datei mit.** Setze Project Options, Linking, "Map file" auf *Detailed* und liefere die `.map` neben der ausführbaren Datei aus. Das

native SDK löst den Trace dagegen auf.

- **Bette die Symbole mit JclDebug ein.** JclDebug faltet die detaillierte Map in das Binary selbst, sodass es nichts Separates zu verteilen gibt. Das SDK liest die eingebettete Info.
- **Nutze madExcept oder EurekaLog wieder, wenn du sie schon hast.** Sie lösen den Stack zur Link-Zeit auf und erzeugen einen vollständig symbolisierten Trace. Eine Ein-Unit-Brücke reicht diesen Trace an ExeWatch weiter, das einen vom Aufrufer gelieferten Stack behält, statt ihn zu ersetzen. Siehe https://exewatch.com/ui/docs#coexisting-madexcept.

Ohne eines davon wird der Absturz zwar aufgezeichnet, aber der Trace bleibt eine Folge von Adressen, mit denen du nichts anfangen kannst. Mach das einmal zum Release-Zeitpunkt und jeder Absturzbericht danach ist lesenswert.

Über die anderen SDKs hinweg ist die Form dieselbe. .NET ist `EW.ErrorWithException(ex, "document")` und erfasst ebenfalls unbehandelte Exceptions automatisch. Python ist `ew.error_with_exception(…)`. Die DLL bietet `ew_ErrorWithStackTrace(Msg, Tag, StackTrace, ExceptionClass)`: Ein flaches C-ABI kann den Stack des Hosts nicht durchlaufen, also löst der Host den Trace auf und übergibt den String. Das JavaScript-SDK läuft nur im Browser (`ew_web_`-Keys) und erfasst `window.onerror` automatisch.



Wie ein erfasster Absturz auf dem Dashboard aussieht: eine `EAccessViolation` mit ihrem Stack, aufgelöst auf Unit, Methode und Zeile, neben der Session, dem Gerät und dem Kunden, die ihn getroffen haben. Das ist ein Build mit Symbol-Info; ohne sie wären dieselben Frames nackte Adressen.

Das letzte Stück ist, gar nicht erst das Dashboard nach Abstürzen absuchen zu müssen. Verdrahte einen Alert auf die Anzahl von `fatal` (behandelt unter Alerts weiter unten) und ein Ausschlag erreicht dich per E-Mail, in Minuten, statt per Support-Ticket, in Tagen.

Rezept #2: Du hast den Stack, aber nicht, was der Benutzer getan hat, um dorthin zu kommen

> Den Fehler kennst du auswendig. Auch die Zeile: `SaveDocument`, eine nil-Referenz, dieselbe `EAccessViolation` seit drei Wochen. Nur passiert er bei genau einem Kunden. Auf deiner Maschine nicht, auf den anderen dreißig Installationen nicht, bei ihm zwei- oder dreimal pro Woche. Du hast keinen Bug zu suchen, du hast eine Reihenfolge zu erraten: etwas, das dieser Benutzer tut und die anderen nicht, das das Programm in einen Zustand bringt, in dem diese Zeile knallt. Du kannst zwei Wochen damit verbringen, ihn zu fragen, was er vorher gemacht hat, und jedes Mal eine andere Rekonstruktion bekommen, weil sich niemand wirklich an seine eigenen Klicks erinnert. Oder du lässt es dir vom Programm erzählen.

Der Stack Trace beantwortet das *Wo*. Breadcrumbs beantworten die andere Hälfte, die dir immer fehlt: **was der Benutzer vorher getan hat**. Es sind Krümel, die du fallen lässt, während die Anwendung arbeitet, und ExeWatch hängt sie von selbst an den nächsten `Error` oder `Fatal`, in Reihenfolge, neben den Stack.

Achte aber darauf, wie sie geschrieben werden, denn hier machen es fast alle beim ersten Mal falsch. Es sind keine fünf Zeilen untereinander: Jede steht da, wo die Sache tatsächlich passiert, verteilt über die Anwendung, und zwischen zweien liegen Bildschirme und Minuten Arbeit des Benutzers.

```
// TInvoiceForm.FormShow -- der Benutzer öffnet eine Rechnung
EW.AddBreadcrumb(btNavigation, 'ui', 'Rechnung geöffnet: ' + IntToStr(AInvoiceId));

// ... hier arbeitet der Benutzer: scrollt die Positionen, korrigiert einen Betrag,
speichert ...

// TUserForm.ChangeRole -- der Benutzer wechselt die Rolle
```

```
EW.AddBreadcrumb(btUser, 'auth', 'Zur Rolle admin gewechselt');

// ... hier macht der Benutzer noch anderes, vielleicht eine Viertelstunde lang ...

// TPriceListImport.Execute -- der Benutzer importiert eine externe Datei
EW.AddBreadcrumb(btFile, 'io',
  Format('%s importiert (%d Zeilen)', [ExtractFileName(AFileName), ARowCount]));

// TInvoiceDAO.ReloadLines -- passiert darunter, der Benutzer sieht es nicht einmal
EW.AddBreadcrumb(btQuery, 'db', 'Rechnungspositionen neu geladen');

// TInvoiceForm.BtnExportClick -- der letzte Schritt vor dem Absturz
EW.AddBreadcrumb(btClick, 'ui', 'Auf Exportieren geklickt');
ExportInvoice(AInvoiceId);   // <-- hier fliegt die Exception, und die fünf Krümel
begleiten sie
```

Achte auch auf die Werte in den Nachrichten: die Rechnungsnummer, der Dateiname, die Zeilenanzahl. In Tags haben diese Werte nie etwas zu suchen, weil jeder unterschiedliche Wert eine eigene Dimension wird und dir das Dashboard sprengt. In einer Breadcrumb-Nachricht sind sie dagegen genau richtig, und dort werden sie erst nützlich, denn zwischen "eine Datei importiert" und "preisliste_2026.csv mit 1284 Zeilen importiert" liegt der ganze Abstand zwischen einer Spur und einer Diagnose.

Wenn dieser Absturz im Dashboard ankommt, liest du nicht mehr, dass es eine Access Violation in `SaveDocument` gibt. Du liest, dass dieser Kunde, und nur dieser Kunde, vor dem Export eine externe Preisliste importiert, und dass dein Export-Code Zeilen dieser Form nie gesehen hat. Die Reihenfolge, die du dir nicht erzählen lassen konntest, steht dort geschrieben.

Dasselbe brauchst du in der schwerer fassbaren Variante des Problems, bei der der Fehler viele Kunden trifft, aber nicht immer. Mit einem Dutzend Abstürzen im Dashboard hörst du auf, in Hypothesen zu denken, und fängst an, die Spuren zu vergleichen: Taucht in allen ein bestimmter Schritt auf und in den gesunden Sessions nie, hast du aufgehört zu raten. Das ist dieselbe Arbeit, die du mit Logs machen würdest, nur ohne vorher wissen zu müssen, welches Log du brauchen wirst.

Der Typ ist einer aus einer festen Menge von sechzehn Werten (`btClick`, `btNavigation`, `btHttp`, `btQuery`, `btUser`, `btForm`, `btFile`, `btState`, `btTransaction`, `btConfig`, `btCustom` und weitere) und gibt dem Dashboard ein Symbol, damit die Spur auf einen Blick lesbar ist. Es gibt auch die Kurzform, `EW.AddBreadcrumb('Export gestartet')`, wenn dir eine Notiz reicht.

Vier Verhaltensweisen, die man kennen sollte, weil sie bestimmen, was du im Ernstfall vor dir hast:

- **Die Spur ist pro Thread, und die Threads sehen einander nicht.** Jeder Thread hält seine eigenen Krümel. Klickt der Benutzer im Hauptthread auf Exportieren und der Fehler fliegt danach in einem Hintergrund-Thread, dann fehlt der Klick in der Spur, die an diesen Fehler

gehängt wird. Wenn du eine asynchrone Arbeit startest, lass auch im Worker einen Krümel liegen, mit dem, was du ihm übergeben hast, sonst reißt die Geschichte genau an der Stelle ab, an der du sie brauchst.

- **Es bleiben die letzten 20 pro Thread.** Der einundzwanzigste verdrängt den ältesten, was an einen Absturz gehängt wird, ist also der unmittelbare Vorlauf und nicht die ganze Session. Deshalb gehören sie nicht in eine Schleife: zwanzig Iterationen `btQuery` füllen die Spur und löschen den Kontext davor.
- **Sie hängen sich nur an `Error` und `Fatal`.** Ein `Info` oder ein `Debug` trägt sie nicht mit, und für sich allein werden sie nie gesendet. Sie verbrauchen also keine Quota, bis ein Fehlschlag sie nutzt.
- **Sie werden verbraucht.** Sind sie einmal an einen Fehler gehängt, wird die Spur dieses Threads geleert, damit der zweite Fehler nicht den Vorlauf des ersten mitschleppt. Das muss man wissen: Schlägt dieselbe Operation zweimal hintereinander fehl, trägt das zweite Event nur die Krümel, die in der Zwischenzeit entstanden sind. Von einem Fehlerpaar hat das erste die vollständige Geschichte.

Was sich als Breadcrumb lohnt: der Wechsel von einem Bildschirm oder Dialog zum nächsten (`btNavigation`, `btForm`), die externen Aufrufe, die du machst (`btHttp`), die Queries, die zählen (`btQuery`), die Aktionen, mit denen der Benutzer den Zustand des Programms ändert, Login, Rollenwechsel, Firmenwechsel (`btUser`), und die Dateien, die er öffnet oder importiert (`btFile`). Die Faustregel ist einfach: Wenn du morgen vor einem Absturz fragen würdest, was der Benutzer vorher gemacht hat, dann ist das ein Krümel, den du heute legst.

Was dir das Dashboard zurückgibt: Du öffnest den Absturz und findest neben dem Stack die letzten zwanzig Schritte, die dorthin geführt haben, in Reihenfolge, mit Typ und Kategorie. Derselbe Absturz von zwei verschiedenen Kunden liest sich wie zwei verschiedene Geschichten, und dort stellt sich meistens heraus, welche davon deine ist.

Rezept #3: Du bist gleich dabei, über ein Feature zu streiten, das vielleicht niemand nutzt

> Ein Planungsmeeting. Jemand ist sicher, dass das Stapel-Export-Feature essenziell ist, und will zwei Wochen, um es zu erweitern. Jemand anderes glaubt, kaum jemand rührt es an. Beide raten, weil keiner eine Zahl hat, und die lauteste Stimme gewinnt diese Art Streit meistens. Es ist genau die Art Frage, die ein einzelner Counter ein für alle Mal klärt.

Setze einen Counter am Einstiegspunkt jedes Features, das dir wichtig ist. Ein Counter ist hier das richtige Werkzeug, kein Log, weil die Frage "wie viele" lautet, und Counter im Backend zu Rollups

summiert werden, sodass du das Gesamt über jede einzelne Installation zurückliest, günstig, ohne dass du eine Zeile pro Klick speicherst.

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

TIP

Inkrementiere einmal pro logischer Nutzung, nicht bei jeder Iteration. Wenn das Erzeugen eines Berichts 500 Zeilen verarbeitet, bleibt es trotzdem eine Nutzung des Features, also ein Inkrement, nicht 500. Würdest du die Zeilen zählen, würdest du die Größe der Berichte messen und nicht, wie oft Leute den Export benutzen, und die zweite war die Frage, die du beantworten wolltest. Interessiert dich auch die erste, ist das ein eigener Gauge: `EW.RecordGauge('report.rows', RowCount, 'reporting')`.

Einen Monat später rankt das Dashboard deine Features nach echter Nutzung, und den Roadmap-Streit entscheiden die Fakten: Du erweiterst, was Leute nutzen, und ziehst still zurück, was sie nicht nutzen.

Rezept #4: Ein Release "fühlt sich langsamer an" und niemand kann es beweisen

> Du lieferst v4.2 aus. Innerhalb einer Woche erwähnen zwei Kunden, der Hauptbericht "wirke seit dem Update träge". Ist das echt, oder ist es der übliche Verdacht, der jeder Änderung folgt? Du kannst es nicht sagen, weil "fühlt sich langsamer an" keine Daten sind, und den Kunden zu bitten, es mit einer Stoppuhr zu messen, ist kein Plan.

Die Lösung ist, die Operation im Feld zu messen und jede Messung mit dem Release zu taggen, das sie erzeugt hat. Umschließe die Operation mit einem Timing-Paar und setze das Release-Label bei init, sodass jede Probe ihre Version kennt:

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

WARNING

Schließe das Paar immer in einem `try..finally` (oder `try..except`). Wenn `RenderReport` eine Exception auslöst, muss `EndTiming` trotzdem laufen, sonst

> bleibt das Timing offen und diese Probe geht verloren, genau auf den Fehlerpfaden, die du am meisten sehen willst.

Das Release-Label kommt aus init: `InitializeExeWatch(ApiKey, CustomerId, '4.2.0')` setzt `AppVersion` für den Lauf, während die Binärversion separat automatisch erkannt wird, du bekommst also beides. Nun der Ablauf, der die Frage beantwortet: Nachdem das Release lange genug im Feld war, um Proben zu sammeln, öffne die "Timing"-Seite, filtere nach Zeitfenster und nutze "Export CSV". Die Datei respektiert deine aktiven Filter und öffnet sich in Excel, wo du Durchschnitts- und p95-Dauer nach `app_version` pivotierst. Die Trägheit hört auf, eine Meinung zu sein, und wird zu "report.render ging in 4.2 von 120ms auf 300ms", was du aus deinen eigenen Felddaten erkannt hast, bevor es sich in Abwanderung verwandelte. Eine native In-UI-Ansicht zum Filtern nach Version ist im Backlog; bis sie kommt, ist der CSV-Pivot der unterstützte Weg, und derselbe Export funktioniert für "Logs", "Timing" und "Metrics" gleichermaßen.

Rezept #5: "Checkout ist langsam", aber wo langsam?

> Kunden sagen, Checkout dauert ewig. Du misst das Ganze und es sind vier Sekunden. Diese Zahl ist für sich fast nutzlos, denn vier Sekunden wovon? Den Warenkorb laden? Das Payment-Gateway? Die Quittung rendern? Wenn du blind optimierst, könntest du einen Tag damit verbringen, die Datenbank-Query schneller zu machen und das Gesamt von vier Sekunden auf drei-Komma-neun bringen, weil die echten Kosten ganz woanders lagen.

Verschachtelte Traces zerlegen diese vier Sekunden. Starte einen Trace, führe ein Timing um jede Phase darin und beende den Trace. Das Backend rekonstruiert ein Wasserfalldiagramm, ein Balken pro Phase, sodass du sehen kannst, welches Kind tatsächlich dominiert:

```
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` liefert eine Trace-ID zurück und `EndTrace` liefert die gesamte verstrichene Zeit in Millisekunden, du kannst das Gesamt also loggen oder darauf assertieren, wenn du willst, und jedes verschachtelte Timing wird zu einem Kind-Span. Auf dem Dashboard wird aus "Checkout ist langsam" ein "Payment ist 80 % des Checkouts", was dir sagt, dass das Problem das Gateway ist, nicht dein Code, und dir den Tag spart, den du mit dem Optimieren der falschen Phase verbracht hättest. .NET und Python spiegeln das exakt; die DLL nutzt `ew_StartTrace(Name, Buffer, BufLen)`, das die Trace-ID in einen vom Aufrufer bereitgestellten Puffer schreibt, und `ew_EndTrace(&ElapsedMs)`.

Ein aggregierter Trace: Jede Phase ist ein Balken mit ihrem Avg, Min, Max und p95, und der Prozentsatz sagt dir, wie viel des Elternteils er ausmacht. Hier dominieren die Transform- und Render-Phasen, also geht genau dorthin die Zeit. Die Statistiken werden über erfolgreiche Läufe berechnet, sodass eine fehlgeschlagene Phase sie nicht verzerrt.

Was passiert, wenn du vergisst, ein Timing zu schließen

Du wirst irgendwann ein `EndTiming` vergessen. Das SDK ist gebaut, um das ohne Speicherleck oder falsche Zahl zu überleben, aber was du bekommst, ist nie so gut wie ein sauberes Paar, und das ist der eigentliche Grund für das try/finally. Hier ist genau, was mit einem offenen Timing passiert, je nachdem, wie es endet:

- **Nichts, wenn es einfach offen bleibt.** Ein Timing ohne passendes `EndTiming` gibt nie eine Probe ab. Es wird nicht gezählt und nicht gemittelt, einfach abwesend. Du verlierst still diese eine Messung.
- **Als fehlgeschlagen automatisch geschlossen, wenn du dieselbe ID erneut startest.** Rufe `StartTiming('report.render')` auf, während ein vorheriges `report.render` auf demselben Thread noch offen ist, und das SDK schließt das alte für dich, markiert mit `success = false` und geflaggt

`auto_closed`, mit einer Warning in deinen Logs ("auto-closed, duplicate StartTiming"). Da es fehlgeschlagen ist, bleibt es aus deinen Dauer-Statistiken; die Warning ist da, um dir zu sagen, dass der Code ein Loch hat.

- **Das älteste wird verdrängt, wenn zu viele auflaufen.** Jeder Thread hält höchstens 100 offene Timings. Starte ein 101., und das älteste offene wird als fehlgeschlagen zwangsgeschlossen, wieder mit einer Warning. Das ist der Auffangposten, der ein langsames Leck vergessener Timings davon abhält, unbegrenzt zu wachsen.
- **Ein Trace räumt seine Kinder auf.** Wenn das vergessene Timing in einem `StartTrace`/`EndTrace` verschachtelt ist, zwangsschließt `EndTrace` jeden Kind-Span, den du offen gelassen hast, als fehlgeschlagen markiert, sodass das Wasserfalldiagramm trotzdem vollständig ist.

Das Muster über alle vier: Ein vergessenes `EndTiming` wird zu einer fehlgeschlagenen Probe plus einer Warning, nie zu einer sauberen Dauer. Das ist die Maschinerie, die dich schützt, kein Feature, auf das man sich stützt. Umschließe jedes Paar in try/finally und nichts davon feuert je.

Rezept #6: Die App ist um 9 Uhr in Ordnung und kriecht um 17 Uhr

> Ein Kunde meldet, dass deine Anwendung morgens flott ist und gegen Ende des Tages träge, und ein Neustart behebt es bis morgen. Dieses Verhalten ist ein Ressourcenleck, und es ist für einen Crash-Reporter unsichtbar, weil nichts abstürzt. Es degradiert einfach, still, über Stunden, auf einer Maschine, die du nie siehst.

Ein Leck ist ein Pegel, der steigt und nicht mehr zurückkommt, und genau das misst ein Gauge. Du willst keine Gauge-Aufrufe durch deine Render-Schleife streuen; du willst, dass der Wert in einem Intervall abgetastet wird. Registriere einen periodischen Gauge und der Sampler-Thread des SDK liest deinen Callback für dich, ohne eigenen Timer:

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

Gute Dinge, die man so beobachtet: Working Set in MB, GDI- oder Handle-Anzahl, offene Dokumentanzahl, Cache-Größe. Nach ein, zwei Tagen zeigt das Dashboard die Signatur deutlich, ein Sägezahn, der jede Session hindurch klettert und beim Neustart abfällt, und weil ein Gauge

Durchschnitt, Min, Max und Probenanzahl pro Fenster behält, sind der steigende Durchschnitt und das Max das Leck. Schneide nach `app_version` und du kannst es oft auf genau das Release festnageln, das es eingeführt hat.

Eine Abweichung, die zu respektieren ist: Der Callback für periodische Gauges existiert auf Delphi, .NET (ein `Func<double>`), Python und JS, aber nicht in der DLL, weil ein Callback das flache C-ABI nicht überqueren kann. Mit der DLL besitzt der Host das Intervall: Betreibe deinen eigenen Timer und rufe `ew_RecordGauge(Name, Value, Tag)` bei jedem Tick auf.

Rezept #7: Sind es alle, oder nur ein Kunde?

> Deine Logs zeigen einen Schwall von Sync-Fehlern und dein erster Instinkt ist, dass die ganze Flotte versagt. Bevor du in Panik gerätst, frage, ob das alle trifft oder nur einen Kunden mit ungewöhnlichem Setup, und beachte, dass das Durchscrollen roher Logs von 200 Installationen dir das nie sagen wird.

Wenn du die Customer-ID bei init setzt und deine Logs nach Subsystem taggst, kannst du jede Scheibe sauber isolieren:

```
EW.SetCustomerId('acme-corp');
// ... später, auf dem Sync-Pfad:
EW.Error('Synchronisation vom Server abgelehnt', 'sync');
```

Nun filtere das Dashboard auf Acmes `sync`-Fehler und das Bild klärt sich in Sekunden: Es ist ein Kunde, hinter einer Unternehmens-Firewall, nicht dein Code, der überall versagt. Geh einen Schritt weiter und erstelle einen Alert, gescoped nach `customer_external_id` und Tag, und du wirst zu Acmes Sync-Problemen speziell benachrichtigt, während du still bleibst über die anderen 200 Kunden, die in Ordnung sind. Fehlerraten pro Kunde und pro Feature, jede für sich alertierbar, alles aus konsistentem Tagging.

Rezept #8: Wie oft schlägt es tatsächlich fehl?

> "Sync schlägt manchmal fehl" ist die Art Bericht, die das Einzige verbirgt, das du wissen musst: Wie oft ist manchmal? Einmal pro Woche ist ein Schulterzucken; jeder dritte Versuch ist ein Vorfall. Du kannst es nicht priorisieren, bis du das Verhältnis sehen kannst.

Die einfachste Version ist ein Counter pro Ergebnis, geschnitten nach einem Outcome-Tag:

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

Aber wenn du die Operation ohnehin schon misst, brauchst du keinen zweiten Counter, weil `EndTiming` ein `success`-Flag als dritten Parameter trägt (Standard `True`):

```
EW.StartTiming('sync', 'sync');
try
  DoSync;
  EW.EndTiming('sync', nil, True);   // erfolgreich
except
  EW.EndTiming('sync', nil, False);  // fehlgeschlagen, aber das Timing schließt
trotzdem
  raise;
end;
```

Dieses Flag tut etwas Feineres als ein Counter, und es lohnt sich, es zu verstehen. Ein fehlgeschlagenes Timing wird nicht verworfen: Es wird aufgezeichnet, sodass du Fehlschläge zählen und sie in der Timing-Liste sehen kannst. Aber es bleibt aus den Dauer-Statistiken heraus. Durchschnitt, Min, Max und p95 auf der "Timing"-Seite werden nur aus erfolgreichen Läufen berechnet. Das zählt, weil Fehlschläge oft die langsamsten Aufrufe sind, eine Operation, die nach einem 30-Sekunden-Timeout aufgab, und wenn die in deine Latenz zählten, würdest du denken, dein Sync sei langsamer geworden, obwohl er in Wahrheit anfing zu scheitern. Mit dem Success-Flag liest du die wahre Latenz der Läufe, die funktionierten, und die Fehlerrate, aus einem Aufruf.

Counter, Gauge oder Timing: welches davon

Diese drei Werkzeuge werden ständig verwechselt, und sie zu verwechseln wirft keinen Fehler, es zeichnet still Daten auf, die nichts bedeuten, was schlimmer ist. Der Unterschied ist einfach, sobald man ihn ausspricht:

Werkzeug	Beantwortet	Wie es aggregiert	Was das falsche kostet
Counter	wie viele, wie oft	über das Fenster summiert	ein Gauge für "Anzahl der Exporte" wirft das Gesamt weg

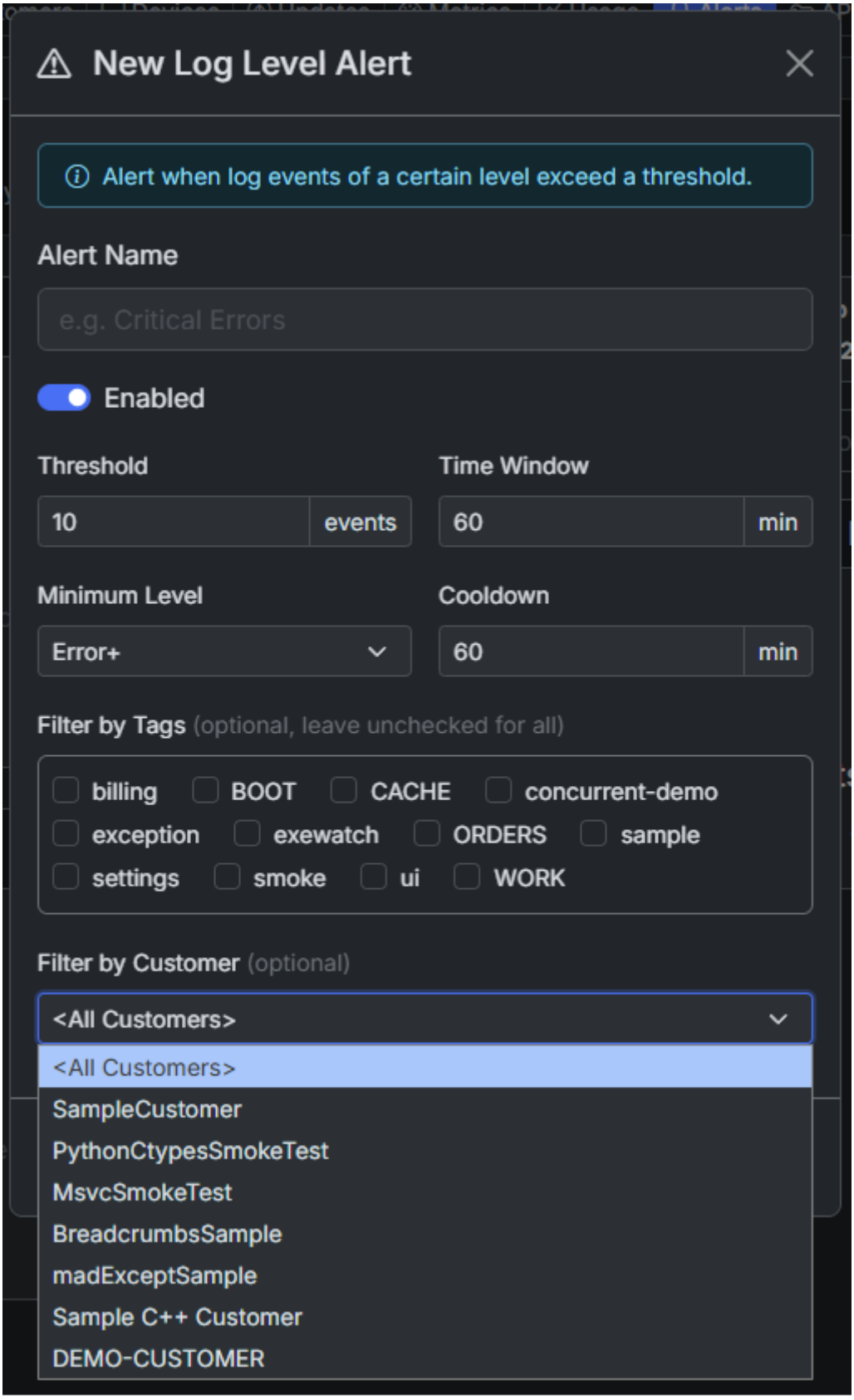
Werkzeug	Beantwortet	Wie es aggregiert	Was das falsche kostet
Gauge	was ist der Wert gerade jetzt	Durchschnitt, Min, Max, Probenanzahl pro Fenster	ein Counter für "Queue-Tiefe" addiert Pegel zu Unsinn
Timing	wie lange hat es gedauert	Dauerverteilung, mit Traces	ein Counter für "wie langsam" gibt dir eine Anzahl, keine Dauer

Im Zweifel addiere zwei Messwerte. Wenn die Summe Sinn ergibt, 12 Exporte plus 8 Exporte sind wirklich 20 Exporte, ist es ein Counter. Wenn sie keinen Sinn ergibt, eine Queue von 12 plus eine Queue von 8 ist keine Queue von 20, ist es ein Gauge. Wenn dich die verstrichene Zeit interessiert, ist es ein Timing.

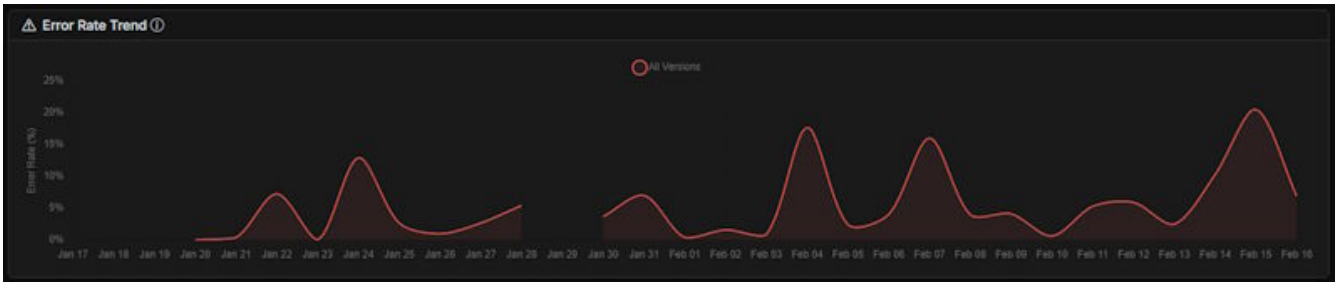
Alerts, die zählen

Ein Alert ist das, was dir erlaubt, das Dashboard nicht mehr beobachten zu müssen. Statt reinzuschauen und zu hoffen, ein Problem zu erwischen, beschreibst du das Problem einmal und ExeWatch mailt dir, wenn es passiert. Zwei Arten gibt es heute, und es lohnt sich, genau zu wissen, worauf jede feuert, denn die ehrlichen Grenzen des Alertings prägen, wie du alles oben nutzt.

Log-Volumen-Alerts feuern, wenn die Anzahl der Log-Events auf oder über einem Level einen Schwellwert innerhalb eines Fensters überschreitet. Du setzt `threshold` (eine Anzahl), `window_minutes`, `min_level` (Standard `error`), einen optionalen `tags`-Filter, eine optionale `customer_external_id` und `cooldown_minutes` (Standard 60), damit ein Vorfall dich nicht fünfzigmal benachrichtigt. Ein typischer lautet: "mehr als 20 `fatal`-Events in 15 Minuten für Kunde Acme."



Derselbe Alert als Formular in der Konsole, unter "Configure alerts": ein Schwellwert über ein Zeitfenster auf einem Mindest-Level, mit einem Cooldown, optional eingegrenzt nach Tag und nach Kunde. "Filter by Customer" auf Acme zu scopen ist der Weg, um über Acmes Fehler benachrichtigt zu werden, ohne von irgendjemand anderem zu hören.



Fehlerrate über die Zeit. Ein Ausschlag wie die hier ist genau das, worauf ein Log-Volumen-Alert achtet, sodass er dich per E-Mail erreicht, sobald er beginnt, statt wenn du zufällig dieses Diagramm öffnest.

Timing-Alerts feuern, wenn Operationen, die auf ein Timing-ID-Muster passen, langsamer als ein Dauer-Schwellwert laufen, oft genug, um einen Anzahl-Schwellwert innerhalb eines Fensters zu überschreiten. Die Felder sind ein `timing_id_pattern`, ein Dauer-Schwellwert in Millisekunden, ein Vorkommen-`threshold` (Standard 5), `window_minutes`, eine optionale `customer_external_id` und `cooldown_minutes` (Standard 240). So wirst du darüber informiert, dass "report.render im Feld langsam läuft", ohne danach zu schauen.

> **NOTE**
>
> Genauso wichtig ist zu wissen, was Alerts heute nicht tun. Es gibt kein "alarmiere, wenn ein Gauge über X geht" oder "alarmiere, wenn ein Counter X überschreitet." Alerts feuern auf Log-Event-Anzahlen und auf Timing, nicht auf beliebige Metrikwerte. Wenn du also benachrichtigt werden willst, sobald Speicher oder Queue-Tiefe eine Linie überschreiten, ist die ehrliche Antwort, dass du das noch nicht kannst; du beobachtest den Gauge auf dem Dashboard. Schwellwert-Alerting auf Metrikwerten ist derzeit kein Feature.

Der Weg, Alerts nützlich statt ignoriert zu halten: Halte deine Log-Level ehrlich (siehe Anti-Patterns, sonst feuert deine `error`-Schwelle auf Dinge, die keine Fehler sind), scope jeden Alert nach Tag oder Kunde, sodass er einen klaren Besitzer hat, und setze ein Cooldown, lang genug, dass ein echter Vorfall als eine E-Mail ankommt.

Versionen mit CSV-Export vergleichen

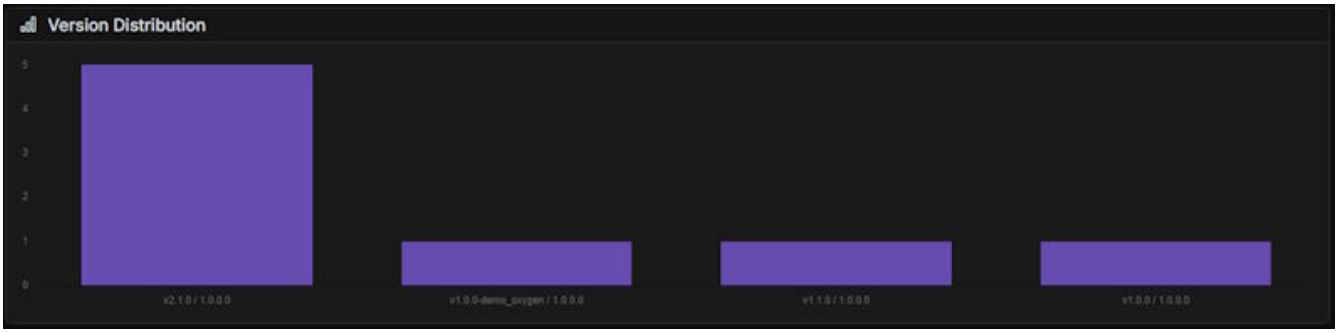
Du lieferst ein Release aus und willst wissen, ob es langsamer oder lauter wurde als das letzte. Heute läuft dieser Vergleich über CSV-Export, und der Ablauf ist es wert, ausbuchstabiert zu werden, weil er eine Frage beantwortet, die Kunden ständig stellen.

1. Liefere das Release mit bei init gesetztem `AppVersion` aus (`InitializeExeWatch(ApiKey, CustomerId, '4.2.0')`). Jedes Event weiß nun, welche Version es erzeugt hat.
2. Gib ihm Zeit im Feld, um Proben zu sammeln, öffne dann die "Timing"-Seite und setze deine

Filter: Zeitfenster, Tag, Kunde.

3. Nutze "Export CSV". Die Datei respektiert diese aktiven Filter und öffnet sich in Excel.
4. Pivotiere Durchschnitts- und p95-Dauer nach `app_version`, und die Regression, falls es eine gibt, steht direkt da in der Pivot-Tabelle.

Derselbe Export existiert auf den "Logs"- und "Metrics"-Seiten, sodass du Fehlervolumen oder Metrik-Pegel über Versionen auf dieselbe Weise vergleichen kannst. Ein In-UI-Timing-Filter pro Version ist im Backlog; wenn er landet, wird daraus ein paar Klicks statt eines Pivots. Bis dahin ist der CSV-Weg der unterstützte Weg, um Version-über-Version-Vergleiche zu machen, und er funktioniert gut genug, dass etliche Teams hier aufhören.



Weil jedes Event seine `AppVersion` trägt, weiß die Konsole bereits, wie deine Releases übers Feld verteilt sind. Genau dieses Versions-Label ist das, wonach du das exportierte Timing pivotierst.

Alles zusammensetzen: eine echte Methode instrumentieren

Isolierte Aufrufe sind leicht abzunicken und schwer einzuordnen. Also hier ist Instrumentierung, hineingelassen in gewöhnlichen Code: eine Sync-Methode auf einem Data Module, die Art, die jede Line-of-Business-Delphi-App hat. Zuerst die schlichte Version, die ihre Arbeit ohne Telemetrie tut:

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

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

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

Nun dieselbe Methode instrumentiert, jede Ergänzung tut genau eine Aufgabe:

```
procedure TSyncModule.SyncInvoices;
var
  Response: IHTTPResponse;
begin
  EW.AddBreadcrumb(btHttp, 'sync', 'GET /invoices since ' + FLastSync);  // Spur für
einen späteren Absturz
  EW.StartTiming('sync.invoices', 'sync');                               // misst die
ganze Operation
  try
    Response := FHttp.Get(FBaseUrl + '/invoices?since=' + FLastSync);
    if Response.StatusCode <> 200 then
      raise ESyncError.CreateFmt('Synchronisation vom Server abgelehnt: HTTP %d',
[Response.StatusCode]);

    EW.StartTiming('sync.import', 'sync');                               // isoliert
die DB-Phase
    FConnection.StartTransaction;
    try
      ImportInvoices(Response.ContentAsString);
      FConnection.Commit;
      EW.EndTiming('sync.import', nil, True);
    except
      FConnection.Rollback;
      EW.EndTiming('sync.import', nil, False);                           // Fehllauf,
raus aus den Statistiken
      raise;
    end;

    FLastSync := NowUtcIso;
    EW.IncrementCounter('sync.completed', 1, 'sync');                    // einer pro
erfolgreichem Sync
    EW.EndTiming('sync.invoices', nil, True);
  except
    on E: Exception do
    begin
      EW.EndTiming('sync.invoices', nil, False);
      EW.ErrorWithException(E, 'sync');                                  // Absturz +
Stack + Breadcrumb
      raise;
    end;
```

```
  end;
end;
```

Was wohin ging, und warum:

- Der **Breadcrumb** steht oben, vor dem Aufruf, den er beschreibt, sodass, wenn irgendetwas weiter unten eine Exception auslöst, die Spur den Request, der dorthin führte, bereits aufgezeichnet hat.
- Das **äußere Timing** (`sync.invoices`) umschließt die ganze Operation; das **innere Timing** (`sync.import`) isoliert die Datenbank-Phase, sodass das Wasserfalldiagramm Netzwerkzeit von Importzeit trennt.
- Der **Server, der schlecht antwortet, löst eine Exception aus**, wie jeder andere Fehler auch: Bei einem Status ungleich 200 gibt es nichts zu importieren, und der Aufrufer muss wissen, dass die Synchronisation nicht stattgefunden hat. Beachte, dass das keine Instrumentierung hinzufügt, sondern welche wegnimmt: Der HTTP-Fehlschlag landet im selben Handler ganz unten, der ihn mit seiner Exception-Klasse und den Breadcrumbs loggt, statt dass ein eigener Zweig sich Logs und Timing-Abschlüsse selbst schreibt.
- Jedes `EndTiming` sitzt sowohl auf dem Erfolgs- als auch auf dem Fehlerpfad und übergibt `False`, wenn die Operation fehlschlägt, sodass ein kaputter Sync deine Latenzzahlen nicht verunreinigt.
- Der **Counter** inkrementiert einmal, nur bei Erfolg, sodass `sync.completed` eine wahre Anzahl guter Syncs ist.
- Der **Fehler** wird mit der Exception am äußersten Handler geloggt, wo er den Stack und den Breadcrumb darüber trägt, dann erneut ausgelöst, sodass die eigene Fehlerbehandlung der App unverändert bleibt.

Beachte die Form: Instrumentierung rahmt den echten Code, sie ersetzt ihn nicht. Lösche jede `EW.`-Zeile und die Methode funktioniert genau wie zuvor. Die Instrumentierung beobachtet nur.

Anti-Patterns

Keines davon heißt "du hast zu viel geloggt". Es sind die Fälle, in denen das, was du schickst, dich in die Irre führt statt dir zu helfen, und der Fix steht neben jedem.

Dieselbe Zeile in einer Schleife wiederholen. Ein `Debug`-Aufruf in einer heißen Schleife oder ein `IncrementCounter` bei jeder Iteration. Es geht nicht um die Menge: Fünfhundert identische Zeilen beantworten alle dieselbe Frage, die schon die erste beantwortet hatte, und verbrauchen dabei die Quota, die die echten Events gebraucht hätten. Zeichne die Operation auf, nicht ihre Iterationen:

eine Zeile beim Start, eine beim Ende oder beim Fehlschlag, und ein Counter-Inkrement pro logischer Operation. Wenn du das Detail pro Iteration während einer Diagnose wirklich brauchst, halte es auf Level `Debug` und hebe das Mindest-Level der Anwendung in der Konsole wieder an, wenn du fertig bist.

Personenbezogene Daten in Logs. E-Mails, Namen, Tokens oder Dateiinhalte in einer Nachricht oder einem Tag. Logs werden aufbewahrt und nach CSV exportiert, und Tags sind niedrig-kardinale Dimensionen, keine Nutzlasten, also ist das sowohl ein Datenschutzproblem als auch ein Durcheinander. Logge IDs und Kategorien; nutze das Customer-ID-Feld für Identität.

Level, die lügen. Alles auf `Error` geloggt, oder echte Fehlschläge auf `Info` geloggt. Alerts haben standardmäßig `min_level = error`, sodass, wenn deine Level falsch sind, deine Alerts mit ihnen falsch sind, und du entweder für nichts benachrichtigt wirst oder nie. Die Skala, die sie ehrlich hält: Fatal heißt, die App kann nicht fortfahren, Error heißt, eine Operation schlug fehl, Warning heißt degradiert, aber funktionierend, Info heißt eine bemerkenswerte Zustandsänderung, Debug heißt nur-Entwickler-Detail.

Gauge und Counter, vertauscht. Ein Counter für "aktuelle Queue-Tiefe" summiert Pegel zu einem bedeutungslosen Gesamt; ein Gauge für "Anzahl der Exporte" wirft das Gesamt weg. Wende den Additionstest aus der Tabelle oben an, bevor du wählst.

Tags, die explodieren. Eine ID, ein Zeitstempel oder irgendein Wert pro Request in einem Tag. Tags sind Dimensionen mit einer kleinen festen Menge von Werten; unbegrenzte Werte vervielfachen sich zu Tausenden einmaliger Dimensionen und machen das Dashboard unbrauchbar. Halte das Vokabular klein.

Erneut loggen, was du schon kostenlos bekommst. Manuell Betriebssystem, Version oder Hardware loggen, die bereits im Device-Snapshot mit jedem Batch mitgeliefert werden. Tu es nicht. Das einzige Feld, das du von Hand setzt, ist `AppVersion`, das Release-Label.

Was passiert, wenn die Internetverbindung abbricht?

Desktop-Anwendungen laufen auf Laptops, die in Tunnel fahren, auf Maschinen hinter wackeligen VPNs, auf einem PC, den jemand zum Feierabend vom Netz trennt. Also ist das eine berechtigte Frage: Was passiert mit deiner Telemetrie, wenn das SDK den Server nicht erreichen kann?

Nichts geht verloren. Das SDK sendet Events nicht direkt von der Aufrufstelle. Es puffert sie, schreibt sie auf die Platte, und ein Hintergrund-Shipper-Thread erledigt das Senden. Wenn das Netz weg ist, schlägt das Senden einfach fehl, die Dateien bleiben auf der Platte, und der Shipper

versucht es in einem Intervall erneut. In dem Moment, in dem die Verbindung zurückkommt, werden die aufgestauten Dateien der Reihe nach verschickt. Ein im Flugzeug geloggter Absturz landet auf deinem Dashboard, sobald der Laptop das nächste Mal online geht.

Vier Einstellungen an `TExeWatchConfig` prägen das:

- `StoragePath` ist der Ort, an dem die ausstehenden Dateien liegen. Standardmäßig ist es ein Ordner pro App; zeige damit auf einen Ort, an den deine App immer schreiben kann.
- `FlushIntervalMs` (Standard 5000) ist, wie oft der In-Memory-Puffer auf die Platte geschrieben wird. Kürzer bedeutet weniger Risiko, falls der Prozess mitten im Lauf beendet wird; länger bedeutet weniger, dafür größere Schreibvorgänge.
- `RetryIntervalMs` (Standard 30000) ist, wie oft der Shipper nach einem Fehlschlag erneut versucht. Das ist deine Reconnect-Taktung: ein niedrigerer Wert baut den Rückstau schneller ab, sobald das Netz zurück ist, um den Preis von mehr Versuchen, solange es noch weg ist.
- `MaxPendingAgeDays` (Standard 7) begrenzt, wie lange ungesendete Dateien aufbewahrt werden. Eine Maschine, die länger als das offline ist, verwirft die ältesten Daten, statt beim Reconnect eine wochenalte Flut zu verschicken. Setze es auf 0, um alles unbegrenzt zu behalten.

Du kannst den Rückstau selbst beobachten: `GetPendingCount` gibt zurück, wie viele Events noch auf den Versand warten.

NOTE

Eine abgebrochene Verbindung ist kein `429`, auch wenn beide damit enden, dass Daten den Server nicht erreichen. Sie sind Gegensätze. Ein Netzausfall ist vorübergehend, also behält das SDK die Daten und versucht es erneut. Ein `429` bedeutet, dass du über der Quota bist, was gewollt ist, also verwirft das SDK die Daten, statt es erneut zu versuchen. Telemetrie an eine schlechte Verbindung zu verlieren, braucht einen Ausfall länger als `MaxPendingAgeDays`; sie an ein `429` zu verlieren, braucht nur das Überschreiten deines Plans, worum es im nächsten Abschnitt geht.

Instrumentiere, was bedeutsam ist, auf dem richtigen Level

Eine technische Tatsache, die man kennen sollte, und sie steht mit Absicht am Ende des Leitfadens: Die monatliche Quota zählt die Events, die du uns schickst, nicht die, die gespeichert bleiben. Es zählen Logs und Metrik-Updates zusammen, also Counter, Gauges und Timings, nicht nur die Logs. Die internen Meldungen des SDK, die mit Tag `ew.system`, fallen nicht in die Zählung. Und weil nur zählt, was angekommen ist, gibt dir Löschen zwar Speicherplatz zurück, aber keine Quota.

Das ist keine Aufforderung, weniger zu instrumentieren. Die richtige Reihenfolge ist die, der du bis hierher gefolgt bist: Zuerst entscheidest du, was du sehen willst, dann steuerst du das Volumen, falls und sobald es ein Thema wird. Die Mittel dafür gibt es, und niemand verlangt von dir, auf Daten zu verzichten, die dich interessieren.

- Das Mindest-Level setzt du pro Anwendung in der Konsole, und das SDK bekommt es bei der nächsten Verbindung. Du kannst mit so viel `Debug` entwickeln, wie du willst, und in Produktion dann nur `Info` und darüber behalten, ohne den Code anzufassen und ohne einen neuen Build auszuliefern.
- Ebenfalls aus der Konsole schickt das Sampling nur einen Bruchteil der Routine-Events. `Error` und `Fatal` umgehen es immer, du kannst also ein gesprächiges Log auf zehntausend Installationen ausdünnen, ohne einen einzigen Absturz zu verlieren.
- Aggregiere, was reine Wiederholung ist: ein Inkrement pro logischer Operation statt eines pro Schleifendurchlauf.
- Nutze Tags zum Schneiden, dann spart dir eine gut getaggte Metrik zehn beinahe gleiche Metriken.

Wenn du dauerhaft nahe an der Grenze läufst und deine Instrumentierung komplett brauchst, gibt es nichts zu kürzen: Du beobachtest mehr Anwendungen oder mehr Kunden als damals, als du den Plan gewählt hast, und dann lohnt sich die nächste Stufe.

NOTE

Über der monatlichen Quota bekommt das SDK ein `429` und beginnt, Daten zu verwerfen, es lohnt sich also, die Nutzung in der Konsole im Blick zu behalten. Nicht um weniger zu loggen, sondern um rechtzeitig zu merken, dass der Plan zu eng wird, statt es daran zu merken, dass du die Events eines Ausschlags verlierst.

Auf der Roadmap

Ein paar Dinge, nach denen Leute fragen, sind geplant, aber nicht ausgeliefert. Sie sind hier ehrlich aufgelistet, damit du weißt, wo heute die Grenzen liegen, und nicht nach einem Feature suchst, das es noch nicht gibt.

- **Erkennen, wenn eine App verstummt.** Ein Totmann-Schalter, sodass du von dem Kunden hörst, dessen App aufgehört hat, sich zu melden, nicht nur von dem, dessen App einen Fehler warf. Alerts feuern heute auf Events, die passieren, nicht auf Events, die aufhören zu passieren, also ist das noch nicht möglich. Geplant.
- **Anomalie-Erkennung.** Automatisches Hervorheben von "das sieht schief aus" auf Metriken, statt dass du feste Schwellwerte setzt. Noch nicht gebaut.

- **Eine Usage-Analytics-Seite.** Eine dedizierte Ansicht für Feature-Adoption und Nutzungsmuster jenseits roher Counter. Noch nicht gebaut.
- **Eine Lese-/Query-API.** Heute sind API-Keys nur zum Ingest, also wartet das Herausziehen deiner Daten für automatisiertes Reporting darauf. Der CSV-Export ist der manuelle Weg in der Zwischenzeit.

Für alles, was es heute gibt, hat die SDK-Referenz in der Konsole die genauen Signaturen, Config-Keys und Installationsschritte. Dieser Leitfaden ist das Was und das Warum; jene ist das Wie.