ExeWatch

Stop guessing. Start knowing.

A DevGuide do ExeWatch

Versão 1.59.0

Adicionar um SDK do ExeWatch a uma aplicação Delphi exige poucas linhas: o Quickstart aqui embaixo já é tudo o que você precisa. Mas assim que a integração está de pé e o dashboard ganha vida, chega a pergunta mais difícil, aquela que o manual de referência não responde: no que vale mesmo a pena ficar de olho?

É a essa pergunta que responde este guia, que fica um degrau acima da referência técnica (a página "Docs" no console, no endereço https://exewatch.com/ui/docs). A referência explica como instalar, inicializar e chamar o SDK; aqui, em vez disso, se fala do que vale a pena chamar, por que e quando. O fio condutor são situações que você reconhece de cara: o chamado de suporte por um crash que ninguém consegue reproduzir, a funcionalidade que você não sabe se alguém usa de verdade, a release que "parece mais lenta" mas ninguém consegue provar.

Quickstart

Uma cláusula `uses` e uma linha de inicialização. Todo o resto do guia não faz nada além de refinar essas duas coisas.

1. Adicione o SDK à sua cláusula uses. Em um app VCL, ou seja, a esmagadora maioria dos apps Delphi, são duas units: a segunda é a que captura as exceções da GUI.

```
uses
  ExeWatchSDKv1, ExeWatchSDKv1.VCL;   // app FireMonkey: ExeWatchSDKv1.FMX no lugar da
.VCL
```

Das duas units de hook você coloca só uma, a do framework que está usando. Se o app não tem GUI, por exemplo um serviço Windows ou um utilitário de linha de comando, não adicione nem a `.VCL` nem a `.FMX`: não existe message loop nenhum onde instalar um hook e `ExeWatchSDKv1` sozinho já basta.

2. Inicialize uma única vez, cedo (no `.dpr` antes de `Application.Run`, ou no evento `OnCreate` do formulário principal):

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

Dois argumentos: a API key que você copia do console e o id do cliente a que pertence esta instalação. Pronto, o app está monitorado.

O que você levou para casa com aquelas duas linhas. Deste momento em diante os crashes são

registrados com o seu stack trace sem que você escreva mais nada, e as duas units dividem o trabalho. `ExeWatchSDKv1` instala um hook em `System.ExceptProc`, onde vão parar as exceções não tratadas do código normal. `ExeWatchSDKv1.VCL` (ou `.FMX`) pega, em vez disso, as que escapam dentro de um evento, um `OnClick` ou um `OnCreate`: quem as intercepta é o message loop do framework, que as entrega a `Application.OnException`, e ali o percurso termina, em `System.ExceptProc` elas nunca chegam. Em um app desktop são a categoria de crash mais frequente, e é por isso que a segunda unit importa.

Sobe também um log automático `Application started`, a confirmação de que a integração está viva, e as informações do dispositivo (sistema operacional, máquina, versão do binário) são enviadas e ligadas ao cliente, para aparecer em "Devices" no console. Nada disso viaja na thread da sua aplicação: os eventos vão para uma fila em disco e uma thread em segundo plano os envia, então uma rede lenta ou um servidor inalcançável não deixam o app lento nem o travam.

A unit de hook não tira de você o handler que já tinha: se você já tinha o seu próprio `Application.OnException`, o hook registra o log e depois o chama; se não tinha, chama `Application.ShowException`. A janela de erro que o seu usuário via antes continua aparecendo igual. E se um dia você esquecer dela em outro projeto, não fica no escuro: o SDK percebe que está rodando em um app com GUI sem hook instalado e escreve ele mesmo um `Warning` com tag `exewatch`, que você encontra no dashboard.

IMPORTANT

O stack trace chega sempre, os números de linha não: para esses é preciso o arquivo `.map`. O compilador Delphi não deixa nomes de unit, método e linha dentro do executável: ele os escreve à parte, em um arquivo `.map` ao lado do binário. O SDK o procura na inicialização com o mesmo nome do executável (`MyApp.exe` → `MyApp.map`) e usa esse arquivo para transformar os endereços em frames legíveis. Se não o encontra, o crash é registrado do mesmo jeito, mas o trace é uma coluna de endereços nus do tipo `[00007FF6A21C3F40]`.

Para tê-los, uma única vez: coloque Project Options, Building, Delphi Compiler, Linking, "Map file" em *Detailed* (os outros níveis, *Segments* e *Publics*, não contêm os números de linha; com *Publics* você obtém os nomes dos métodos mas não as linhas). Depois distribua o `.map` junto com o executável, na mesma pasta.

Não pode distribuí-lo? É uma escolha legítima: o `.map` é um arquivo de texto de alguns megabytes que expõe os nomes internos do seu código. Nesse caso arquive o `.map` de cada release junto com o binário que você entregou, porque os endereços que você vê no dashboard continuam resolvíveis mais tarde, mas só com o map daquela build idêntica: recompilar os desloca e o

> map novo não serve para nada. Quem não quer gerenciar arquivos separados tem outros dois caminhos: JclDebug, que embute os símbolos dentro do próprio executável, e madExcept ou EurekaLog se você já os usa, que resolvem o stack na fase de link e entregam ao ExeWatch um trace já simbolizado. Você encontra os dois mais adiante, na nota da Receita #1.

3. Registre o que te interessa. Daqui em diante você registra, conta e cronometra toda vez que tiver algo digno de nota:

```
EW.Info('Relatório mensal gerado', 'reporting');
EW.IncrementCounter('report.generated', 1, 'reporting');
```

Abra o dashboard e os eventos estão lá. Daqui para a frente o guia serve para te fazer lembrar de tudo o que merece acabar lá dentro, porque a lista é mais longa do que a que te viria à cabeça agora.

Depois, uma configuração de cada vez

Nada do que vem a seguir é necessário para começar: acrescente quando precisar de verdade, uma linha por vez.

Quer saber de qual release vem um evento? Passe um terceiro argumento.

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

É o `AppVersion`, o seu rótulo de release ('4.2.0', '2026-Q1', 'v2-beta'). A versão do binário é um campo à parte, que o SDK já lê sozinho do executável: este terceiro argumento serve quando a sua ideia de release não coincide com o número compilado no `.exe`.

Na inicialização você ainda não sabe quem é o cliente? Inicialize do mesmo jeito, com id vazio, e defina o id assim que descobrir. Como e por que está na seção *Inicialização: além da linha única*, logo mais adiante.

Integrar cada SDK

O Quickstart era em Delphi, o SDK carro-chefe. O *o quê* e o *por quê* no resto do guia valem igualmente para cada SDK: muda apenas o modo de integrá-lo e inicializá-lo. Eis o setup de cada um. Em todos o último argumento é o rótulo de release (`AppVersion`), e a API key tem o prefixo da plataforma (`ew_win_`, `ew_lin_`, `ew_web_` e assim por diante).

Delphi (nativo). Como no Quickstart: `ExeWatchSDKv1` na cláusula uses, mais `ExeWatchSDKv1.VCL` (ou `.FMX`) se o app tem GUI, depois `InitializeExeWatch(ApiKey, CustomerId, '4.2.0')`. A unit VCL/FMX captura para você as exceções de GUI não tratadas.

.NET. Referencie o pacote `ExeWatch` (adicione `ExeWatch.WinForms` para um app WinForms), depois inicialize uma única vez ao iniciar:

```
using ExeWatch;

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

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

Um app console ou um serviço pula a linha `Install`: o client instala sozinho um hook em `AppDomain.UnhandledException`. Já o WinForms exige a chamada `Install` explícita antes de `Application.Run`.

Python. Instale o SDK, inicialize o singleton e depois use o objeto `ew` em nível de módulo:

```
from exewatch import initialize_exewatch, ew

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

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

Python não tem uma GUI onde instalar um hook, então capture as falhas onde você as trata com `ew.error_with_exception(exc, "tag")`, ou aponte `sys.excepthook` para o SDK para ter uma rede de segurança global.

JavaScript (browser). Você declara a configuração em `window.ewConfig` e depois carrega o script de exewatch.com. A chave é uma chave `ew_web_`, e o SDK captura automaticamente `window.onerror`:

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

<!-- ...depois o script, servido por exewatch.com -->
<!-- Produção (minificado, 12 KB) -->
```

```
<script src="https://exewatch.com/static/js/exewatch.v1.min.js"></script>

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

A ordem importa: o SDK se inicializa sozinho no `DOMContentLoaded` lendo `window.ewConfig`, portanto a configuração já deve estar na página quando o script for carregado. O script você serve de exewatch.com, não de uma cópia sua, assim as correções chegam aos seus usuários sem que você tenha que redistribuir nada.

Este SDK é só para browser, não existe uma build para Node. Os erros não capturados são coletados para você, junto com as falhas de `fetch` e `XHR` e os `console.error`. Em uma página que carrega scripts de terceiros, `ignoreUrls` na mesma `ewConfig` descarta os erros que vêm de domínios que você não controla, tipo publicidade e analytics: é ruído que não te diz nada sobre a sua aplicação.

DLL (C, C++, VB, .NET legado, qualquer coisa com uma ABI C). Carregue `ExeWatchSDKv1DLL.dll` e chame os exports flat. As strings são wide (`PWideChar`), cada função é `stdcall` e os resultados voltam como códigos de retorno:

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

Como uma ABI C flat não consegue percorrer o stack do host nem executar um callback, duas tarefas recaem sobre o host: passar uma string de stack já resolvida para `ew_ErrorWithStackTrace`, e disparar os gauges periódicos a partir de um timer seu (não existe callback para os gauges periódicos). Um host Delphi que prefira a DLL às units nativas pode usar a import unit `ExeWatchSDKv1Imports` e chamar `EWInitialize(…)` no lugar de `InitializeExeWatch`.

O mesmo, por inteiro, com MSVC. Nenhuma import library e nenhum runtime Embarcadero: defina `EW_DYNAMIC_LOAD` antes do header e cada `ew_*` vira um ponteiro de função, que `ExeWatchSDKv1.dynload.c` resolve em runtime com `LoadLibrary` e `GetProcAddress`. Este é um programa completo, não um fragmento:

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

int wmain()
{
    // procura ExeWatchSDKv1DLL_x64.dll no path de busca padrão do Windows
    if (ew_LoadSDK() != EW_OK)
    {
        fwprintf(stderr, L"ExeWatchSDKv1DLL_x64.dll não encontrada\n");
        return 1;
```

```
    }

    if (ew_Initialize(L"ew_win_xxxxxxxx", L"acme-corp", L"4.2.0") != EW_OK)
    {
        wchar_t err[1024] = {};
        ew_GetLastError(err, 1024);
        fwprintf(stderr, L"Falha no Init: %ls\n", err);
        ew_UnloadSDK();
        return 1;
    }

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

    ew_WaitForSending(15);   // retorna quantos eventos restam na fila: 0 = tudo
enviado
    ew_Shutdown();
    ew_UnloadSDK();
    return 0;
}
```

Compila-se em um comando só, a partir de um prompt "x64 Native Tools Command Prompt for VS 2022", passando a pasta que contém header e loader:

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

A única linha que aqui não é cerimônia é `ew_WaitForSending`. Um utilitário de linha de comando pode terminar antes que a thread de envio tenha esvaziado a fila, e aquela chamada escreve o buffer em disco e espera a fila esvaziar, retornando quantos eventos ainda estão aguardando. Não é um problema de perda de dados, porque a fila fica em disco e recomeça na execução seguinte, mas sem a espera os seus eventos aparecem no dashboard com uma rodada de atraso. O exemplo compilável com todo o resto, identidade do usuário, tags globais, breadcrumbs, timings aninhados e gauges, está em ExeWatchSamples/MSVCWithDLLSDK.

A mesma DLL a partir de um console Delphi. Um host Delphi também pode usar a DLL em vez das units nativas, e em dois casos é a escolha certa: quando você está em um Delphi mais antigo que o XE8, que é o mínimo do SDK nativo, e quando você tem várias aplicações que precisa atualizar distribuindo um binário só. A import unit `ExeWatchSDKv1Imports` chega até o Delphi 5. Este também é um programa inteiro:

```
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('Falha no Init: ', EWGetLastErrorStr);
    Exit;
  end;

  EWInfo('Batch noturno iniciado', 'batch');
  EWIncrementCounter('report.generated', 1.0, 'reporting');

  LRemaining := ew_WaitForSending(15);
  if LRemaining > 0 then
    WriteLn('Restam ', LRemaining, ' eventos na fila: vão sair na próxima execução.');

  ew_Shutdown;
end.
```

Os wrappers com o prefixo `EW` aceitam `string` e a conversão para `PWideChar` eles fazem sozinhos, portanto no seu código não aparece nenhum cast. Por se tratar de um console, valem as mesmas duas linhas de antes: `ew_WaitForSending` antes de sair, e `ew_Shutdown` para fechar limpo.

Uma coisa para saber antes de distribuir. Por padrão a unit liga a DLL estaticamente, portanto `ExeWatchSDKv1DLL_x64.dll` precisa estar ao lado do executável no momento da inicialização: se faltar, o processo não sobe de jeito nenhum, o Windows o bloqueia com um erro de DLL ausente antes mesmo da sua primeira linha de código. Se você prefere que a aplicação suba de qualquer forma e abra mão apenas da telemetria, defina `EW_DYNAMIC_LOAD` nas opções de projeto: a unit passa a usar `LoadLibrary`, e você chama `EWLoadDLL` na inicialização verificando o resultado, como faz o exemplo MSVC aqui em cima.

E com Free Pascal. A DLL não tem nada de específico do Delphi: é uma ABI C flat, `stdcall`, strings wide. No Windows a mesma `ExeWatchSDKv1Imports` compila com o FPC, que a unit reconhece e coloca em `{$MODE DELPHI}` sozinha. Onde `string` não é Unicode o alias interno vira `WideString` e os wrappers `EW*` convertem para você, portanto o código que você escreve continua sendo o do exemplo aqui em cima.

Inicialização: além da linha única

A única chamada a `InitializeExeWatch` do Quickstart é tudo de que a maioria dos apps vai precisar algum dia. Aquele terceiro argumento é o `AppVersion`, o seu rótulo de release ('4.2.0', '2026-Q1', 'v2-

beta'); a versão do binário, por sua vez, é um campo separado, detectado automaticamente a partir do executável, portanto daquela única linha você obtém as duas. Eis o que fazer quando aquela linha não bastar.

> **TIP**
>
> **Ainda não sabe quem é o cliente? Inicialize do mesmo jeito.** Na maioria dos apps reais o SDK deveria estar ativo desde a primeira linha, para que um crash durante a inicialização seja capturado, mas você só descobre a que cliente pertence a instalação depois de ler um arquivo de licença ou depois que o usuário fez login. Inicialize com um id de cliente vazio e defina o id assim que souber qual é.

```
// no .dpr, antes de Application.Run, assim o SDK fica ativo desde já
InitializeExeWatch('ew_win_xxxxxxxx', '', '4.2.0');

// ... mais adiante, quando você souber quem é o cliente (de um arquivo de licença ou
depois do login):
EW.SetCustomerId(Session.CustomerCode);
```

É um fluxo deliberado e suportado, não um improviso. Enquanto o id do cliente estiver vazio o SDK segura o registro com as informações do dispositivo, porque ele precisa do id para ligar o dispositivo ao cliente certo, e o envia no instante em que você chamar `SetCustomerId`. Se depois o id mudar (um cliente diferente na mesma máquina) o SDK reenvia as informações do dispositivo sob o novo cliente, assim o dispositivo aparece corretamente para os dois. A regra é simples: primeiro inicialize, depois identifique assim que puder.

Cliente e usuário são duas identidades diferentes, definidas com duas chamadas diferentes. `SetCustomerId` define o *cliente*: a conta ou tenant a que pertence esta instalação, ou seja, o critério com que o dashboard agrupa dispositivos e alertas. Quem está de fato usando o app é outra coisa, o *usuário*, e esse você define com `SetUser`:

```
// depois que a pessoa fez login:
EW.SetUser('u-8842', 'mario.rossi@acme.example', 'Mario Rossi');
// no logout:
EW.ClearUser;
```

Id, e-mail e nome viajam depois com cada evento como `user_id`, assim um crash mostra não só qual cliente o encontrou mas também qual pessoa. Use `SetCustomerId` para a conta e `SetUser` para a pessoa: são independentes, e você pode definir um, o outro, os dois ou nenhum dos dois.

> **WARNING**
>
> **Tanto `SetCustomerId` quanto `SetUser` valem para o processo inteiro: servem para um app single-user, não para um servidor multiusuário.** Cada um deles é um único valor na instância do SDK durante todo o processo, não algo

> ligado a uma thread ou a uma requisição. É exatamente o que serve para uma aplicação desktop, onde um só cliente e um só usuário autenticado estão ativos por vez. Já é errado para um app do lado do servidor que atende muitos usuários juntos: duas requisições em duas threads sobrescreveriam uma a identidade da outra, e os eventos seriam atribuídos a quem a definiu por último. O ExeWatch é pensado para aplicações desktop e single-user. Em um servidor multi-tenant não rastreie a identidade por requisição desse jeito; leve-a para dentro do próprio evento (uma tag por chamada ou um campo `extra_data`).

Quando a chamada de três argumentos não bastar, construa um `TExeWatchConfig` e passe-o. Cada campo tem um padrão sensato, então defina só o que você precisa:

```
var
  Config: TExeWatchConfig;
begin
  Config := TExeWatchConfig.Create('ew_win_xxxxxxxx', 'acme-corp');
  Config.AppVersion := '4.2.0';
  Config.SampleRate := 0.25;                // envia 25% dos eventos de rotina;
Error/Fatal passam sempre
  Config.GaugeSamplingIntervalSec := 60;    // de quanto em quanto lê os gauges
periódicos (padrão 30, mínimo 10)
  Config.MaxPendingAgeDays := 3;            // descarta os arquivos na fila mais
antigos que isso (padrão 7)
  Config.AnonymizeDeviceId := True;         // substitui por um hash o nome de usuário
no id do dispositivo (GDPR / AD)
  Config.GlobalTags := [TPair<string, string>.Create('edition', 'pro')];
  InitializeExeWatch(Config);
end;
```

As configurações que vale a pena conhecer:

- **SampleRate** (de 0 a 1): a fração dos eventos de rotina efetivamente enviados. `Error` e `Fatal` sempre passam por cima da amostragem, assim você pode ralear o volume de info e debug em um parque de máquinas grande sem nunca perder um crash.
- **GlobalTags** e **InitialCustomDeviceInfo**: tags e campos de dispositivo aplicados antes mesmo do primeiríssimo evento, de modo que até o log automático "Application started" já carregue consigo o seu contexto.
- **GaugeSamplingIntervalSec**: de quanto em quanto a thread de amostragem lê os seus gauges periódicos (padrão 30s, mínimo 10s).
- **MaxPendingAgeDays**: enquanto o app está offline, os eventos se acumulam em disco; os arquivos mais antigos que esse valor são eliminados, assim uma máquina que ficou offline por

semanas não envia dados velhos na reconexão (padrão 7).

- **AnonymizeDeviceId**: substitui por um hash a parte de nome de usuário do id do dispositivo, para ambientes GDPR ou Active Directory.
- **Endpoint**: faz o SDK apontar para uma instância ExeWatch self-hosted em vez da nuvem padrão. Somente on-premise.

Os outros SDKs adotam os mesmos conceitos sob nomes de campo bem parecidos; a referência tem a assinatura exata de cada um.

Por que este guia existe

Um dashboard vazio intimida, e a primeira pergunta é sempre a mesma: por onde eu começo? A resposta curta é que se uma coisa te interessa, ela precisa ser registrada. Te interessa saber se aquela função falha? Registre. Quantas vezes ela é usada? Conte. Se é lenta e quanto? Cronometre. O erro que se paga caro não é ter mandado alguns eventos a mais, é se ver diante de um problema em produção e descobrir que justamente aquele pedaço de código não conta nada.

Aquele momento sempre chega, e chega numa terça à tarde com um cliente no telefone. Você tem o stack trace do crash mas não sabe o que o usuário estava fazendo um instante antes, porque aquela passagem você não tinha registrado. Sabe que a sincronização demora uma eternidade mas não qual fase a devora, porque você tinha cronometrado só o total. Naqueles vinte minutos você nunca lamenta as linhas que mandou a mais. Você lamenta as três que não escreveu.

Portanto comece generoso. Se algo pode falhar, se pode ficar lento, se você precisa saber o quanto é usado, se explica por que o app se comportou de um certo jeito, mande. Acrescentar é fácil enquanto você está escrevendo o código; acrescentar depois, quando a build em questão já está instalada em trezentos clientes, significa um lançamento e semanas de espera. E o ruído, se um dia houver demais, você tira quando quiser: o nível mínimo e a amostragem de cada aplicação você muda pelo console, sem recompilar nada.

A única coisa que não vale a pena mandar é aquela que não diz nada nem para você: um `Debug('estou aqui')`, um `Info('step 1')`, uma mensagem que te parece útil enquanto você a escreve só porque naquele momento o contexto está na sua cabeça. Quando você a reencontra em uma lista de logs, meses depois, aquele contexto não existe mais e a linha perdeu todo o sentido. O resto do guia serve para te fazer lembrar do que, esse sim, merece estar ali, e para escolher a ferramenta adequada a cada coisa, visto que um crash, uma contagem de uso e uma duração se registram de três formas diferentes.

As cinco perguntas

Antes das receitas, o mapa. O ExeWatch coloca à sua disposição cinco ferramentas, e cada uma existe para responder a uma pergunta diferente. Quase toda decisão do tipo "o que eu rastreio aqui?" fica óbvia assim que você sabe qual pergunta está se fazendo.

A pergunta que você tem	A ferramenta que responde a ela	Em que você a aponta
O que aconteceu?	Log (de debug a fatal, com tag e stack trace)	eventos discretos que uma pessoa gostaria de ler: erros, mudanças de estado, "o usuário fez X"
Quantos, com que frequência?	Counter	coisas que você conta e soma: uso de funcionalidades, retries, exports, falhas
Qual é o valor neste momento?	Gauge	um nível que sobe e desce: memória em MB, profundidade de fila, documentos abertos, tamanho da cache
Quanto tempo demorou?	Timing (e traces aninhados)	durações de operações, com trace para decompor uma lenta nas suas fases
Me avise quando algo der errado	Alerts	um limiar sobre o volume de logs error ou fatal, ou sobre a duração de uma operação, configurado no console, não no código

Duas coisas escapam com facilidade, e as duas te poupam trabalho de verdade mais adiante.

As tags são a sexta ferramenta silenciosa. Toda chamada de log, counter, gauge e timing aceita uma `tag`, e existe um `SetTag` válido para o processo inteiro, para o contexto que viaja com tudo. As tags são o modo como você fatia o dashboard em "payment" contra "sync" contra "ui" sem inventar uma métrica separada para cada funcionalidade. Acerte nelas e todo gráfico é filtrável exatamente como você raciocina; erre nelas e o dashboard vira ruído. A seção sobre higiene das tags é a próxima, e é curta.

As informações do dispositivo são automáticas e de graça. Versão do app, versão do binário, OS e hardware viajam com cada lote sem uma única chamada da sua parte. Não os registre de novo. O

único campo que você define na mão é o `AppVersion`, o rótulo de release, e é justamente ele que mais adiante torna possível a comparação entre versões.

Higiene e nomes das tags

Uma tag serve para filtrar, não para transportar dados. Mantenha-as curtas, poucas e estáveis ao longo do tempo, e o dashboard continua legível mesmo depois de alguns anos de acréscimos.

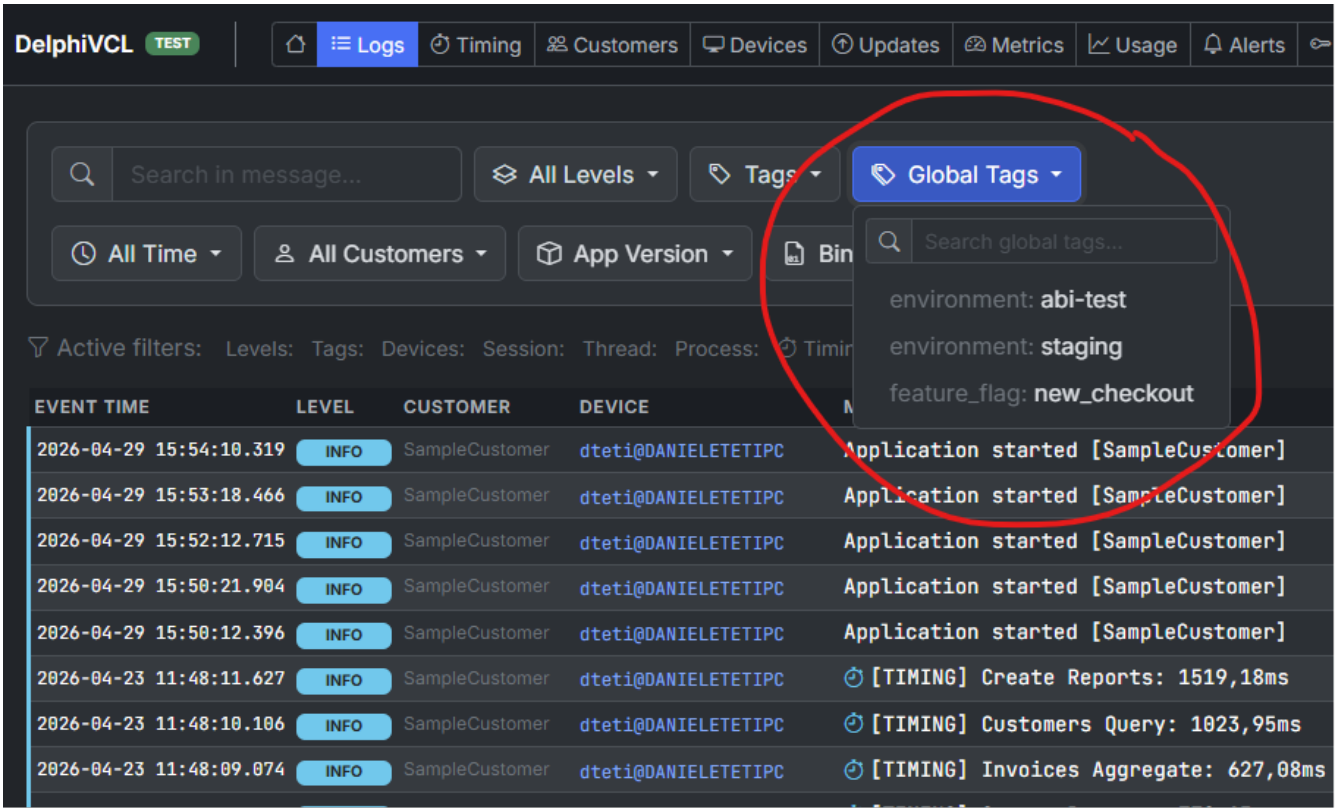
Na prática:

- Use um vocabulário pequeno e fixo: `payment`, `sync`, `ui`, `export`, `startup`. Um punhado de tags cobre a maioria das aplicações. Se você perceber que está inventando uma tag nova toda semana, pare.
- Nomeie counters e ids de timing no estilo `feature.operation`: `report.render`, `invoice.export`, `sync.retry`. Os prefixos compartilhados se agrupam sozinhos no dashboard, assim todos os seus números `sync.*` ficam juntos sem configuração nenhuma.
- Não coloque em uma tag um id, um timestamp, um nome de arquivo ou um valor que muda a cada chamada. Cada valor distinto vira uma dimensão à parte, então uma tag com o número do pedido dentro cria milhares delas e o filtro não serve mais para nada. Isso se chama alta cardinalidade. Se você precisa daquele valor, o lugar certo é a mensagem do log ou um breadcrumb.
- Nada de dados pessoais nas tags nem nas mensagens de log, que são conservadas e exportadas em CSV. Registre ids e resultados, não e-mails, nomes, tokens ou conteúdos de arquivos. Para a identidade existe o campo id do cliente.

Defina uma única vez, na inicialização, o contexto que nunca muda, assim você não o repete a cada chamada:

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

A partir daquele momento cada evento carrega consigo aquele contexto, e você pode filtrar o dashboard inteiro até chegar, por exemplo, apenas à edição Pro no canal stable sem tocar em outra linha de código.



As tags que você define com `SetTag` (ou com `GlobalTags` no init) viram um filtro "Global Tags" no console, assim você pode fatiar cada visão por ambiente, edição, feature flag e qualquer outra coisa que você tenha etiquetado.

As mesmas chamadas, em cada SDK

As receitas que seguem usam Delphi. Os conceitos são idênticos entre os SDKs; muda só a grafia. Esta tabela é o mapa, assim um leitor .NET ou Python consegue traduzir qualquer snippet num relance. A referência no console tem as assinaturas completas.

O que você quer	Delphi (nativo)	.NET (`EW`)	Python (`ew`)	JavaScript (`ew`)	DLL (C flat)
Registrar um erro com stack	`EW.ErrorWithException(E, 'tag')`	`EW.ErrorWithException(ex, "tag")`	`ew.error_with_exception(exc, "tag")`	`ew.error('msg', 'tag')`	`ew_ErrorWithStackTrace(…)`
Registrar em um nível	`EW.Info('msg', 'tag')`	`EW.Info("msg", "tag")`	`ew.info("msg", "tag")`	`ew.info('msg', 'tag')`	`ew_Log(level, …)`
Contar alguma coisa	`EW.IncrementCounter('x', 1, 'tag')`	`EW.IncrementCounter("x", 1, "tag")`	`ew.increment_counter("x", 1, "tag")`	`ew.incrementCounter('x', 1, 'tag')`	`ew_IncrementCounter(…)`

O que você quer	Delphi (nativo)	.NET (`EW`)	Python (`ew`)	JavaScript (`ew`)	DLL (C flat)
Registrar um gauge	`EW.RecordGauge('x', v, 'tag')`	`EW.RecordGauge("x", v, "tag")`	`ew.record_gauge("x", v, "tag")`	`ew.recordGauge('x', v, 'tag')`	`ew_RecordGauge(…)`
Gauge periódico	`EW.RegisterPeriodicGauge(…)`	`EW.RegisterPeriodicGauge(…)`	`ew.register_periodic_gauge(…)`	`ew.registerPeriodicGauge(…)`	*(nenhum: poll + `ew_RecordGauge`)*
Cronometrar uma operação	`EW.StartTiming` / `EndTiming`	`EW.StartTiming` / `EndTiming`	`ew.start_timing` / `end_timing`	`ew.startTiming` / `endTiming`	`ew_StartTiming` / `ew_EndTiming`
Trace aninhado	`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(…)`
Definir uma tag	`EW.SetTag('k', 'v')`	`EW.SetTag("k", "v")`	`ew.set_tag("k", "v")`	`ew.setTag('k', 'v')`	`ew_SetTag(…)`
Definir o id do cliente	`EW.SetCustomerId('id')`	`EW.SetCustomerId("id")`	`ew.set_customer_id("id")`	`ew.setCustomerId('id')`	`ew_SetCustomerId(…)`
Definir o usuário atual	`EW.SetUser('id', 'email', 'name')`	`EW.SetUser("id", "email", "name")`	`ew.set_user("id", "email", "name")`	`ew.setUser({id, email})`	`ew_SetUser(…)`

Dois fatos específicos dos SDKs para levar consigo em cada receita: a **DLL não tem callback para os gauges periódicos**, então a amostragem você dispara a partir de um timer seu, e o **JavaScript é só para browser**. Todo o resto é uma simples reescrita um para um.

Receitas

Cada receita parte de uma situação em que você já se viu, ou vai se ver, e sobe até a única coisa que vale a pena instrumentar. Um snippet Delphi canônico para cada uma; use a tabela aqui em cima para traduzi-lo para o seu SDK. Onde o comportamento muda de verdade, está sinalizado.

Receita #1: Um cliente diz que deu crash, e você não faz ideia do porquê

> O ticket de suporte diz: "o programa fechou sozinho e eu perdi meu trabalho". Nenhum passo, nenhum screenshot, nenhum número de versão.

> Sem telemetria a sua única jogada é pedir ao cliente que reproduza um crash que ele não consegue reproduzir sob demanda, em uma máquina que você não pode ver. Metade das vezes o ticket morre ali, sem solução, e o bug fica em campo.

É exatamente a situação para a qual existem os logs e o stack trace, e a boa notícia é que ativá-los é uma linha de setup. Adicione `ExeWatchSDKv1.VCL` (para um app VCL) ou `ExeWatchSDKv1.FMX` (para FMX) à cláusula uses, e o SDK instala para você um hook no caminho das exceções do framework: cada exceção de GUI não tratada é capturada como `Fatal`, etiquetada `exception`, com o seu stack trace, e descarregada na hora para que nada se perca enquanto o app morre. Se você esquecer a unit e o app for uma GUI, o SDK percebe e te avisa para adicioná-la. Quando o cliente levantar o telefone, o crash já está no seu dashboard, com o stack, a versão do app, o OS e o id do cliente anexados. Você não está mais pedindo que reproduzam nada: está lendo o que aconteceu.

As exceções que você mesmo captura e trata não passam por aquele hook, então essas você registra explicitamente com a sobrecarga de exceção, que leva consigo o stack trace:

```
try
  ProcessDocument(ADoc);
except
  on E: Exception do
  begin
    EW.ErrorWithException(E, 'document');
    // trate ou relance conforme o seu app precisar
  end;
end;
```

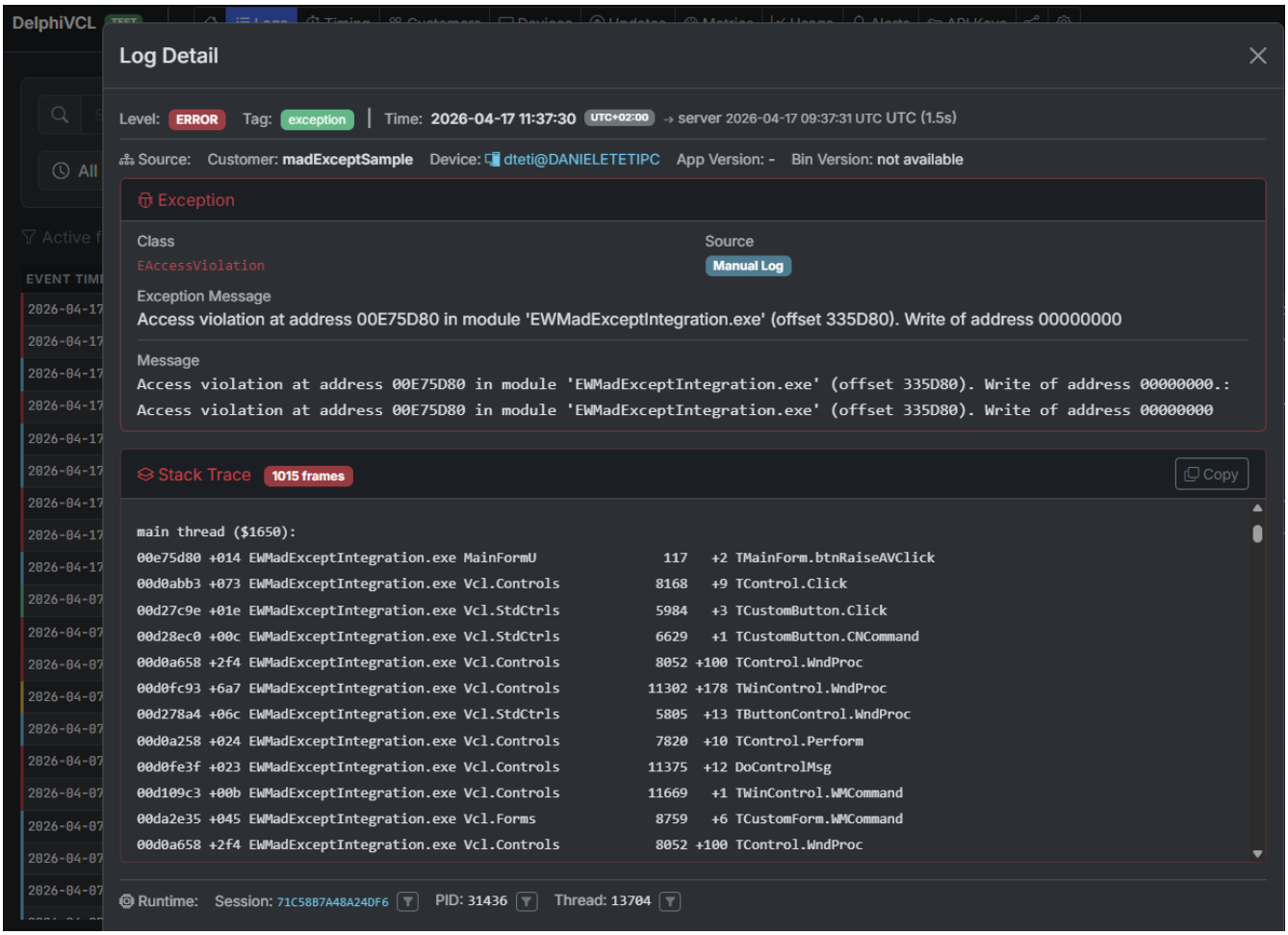
NOTE

Um stack trace só serve se você souber lê-lo. Capturado cru, é uma lista de endereços de memória, não nomes de método e números de linha. Para obter frames legíveis você precisa entregar as informações de debug a qualquer coisa que resolva o stack. Três modos, escolha o adequado à sua build:

- **Distribua um arquivo MAP detalhado.** Coloque Project Options, Linking, "Map file" em *Detailed*, e distribua o `.map` ao lado do executável. O SDK nativo resolve o trace comparando-o com aquele arquivo.
- **Incorpore os símbolos com o JclDebug.** O JclDebug embute o map detalhado dentro do próprio binário, assim não há nada separado para distribuir. O SDK lê as informações incorporadas.
- **Reuse madExcept ou EurekaLog se você já os tem.** Eles resolvem o stack na fase de link e produzem um trace plenamente simbolizado. Uma ponte de uma

> única unit passa aquele trace ao ExeWatch, que mantém o stack fornecido por quem chamou em vez de substituí-lo. Veja https://exewatch.com/ui/docs#coexisting-madexcept.
>
> Sem nenhum desses o crash é registrado do mesmo jeito, mas o trace continua sendo uma sequência de endereços sobre a qual você não pode agir. Faça isso uma vez no momento do lançamento e todo relatório de crash seguinte passa a ser digno de leitura.

Nos outros SDKs a forma é a mesma. .NET é `EW.ErrorWithException(ex, "document")` e também captura automaticamente as exceções não tratadas. Python é `ew.error_with_exception(…)`. A DLL expõe `ew_ErrorWithStackTrace(Msg, Tag, StackTrace, ExceptionClass)`: uma ABI C flat não consegue percorrer o stack do host, então é o host que resolve o trace e passa a string. O SDK JavaScript é só browser (chaves `ew_web_`) e captura automaticamente `window.onerror`.



Qual é a cara de um crash capturado no dashboard: um `EAccessViolation` com o seu stack resolvido em unit, método e linha, ao lado da sessão, do dispositivo e do cliente que o encontraram. Esta é uma build com informações de símbolo; sem elas, os mesmos frames seriam endereços nus.

A última peça é não precisar nem olhar o dashboard atrás de crashes. Configure um alerta sobre a contagem de `fatal` (tratado mais adiante em Alertas) e um pico te alcança por e-mail, em poucos

minutos, em vez de por ticket de suporte, em dias.

Receita #2: Você tem o stack, mas não o que o usuário fez para chegar lá

> O erro você conhece de cor. Sabe até a linha: `SaveDocument`, uma referência nil, a mesma `EAccessViolation` há três semanas. O ponto é que só acontece em um cliente. Na sua máquina não, nas outras trinta instalações não, na dele duas ou três vezes por semana. Você não tem um bug para procurar, tem uma sequência para adivinhar: algo que aquele usuário faz e os outros não, que leva o programa a um estado em que aquela linha explode. Você pode passar duas semanas perguntando a ele "mas antes o que você estava fazendo?" e receber toda vez uma reconstrução diferente, porque ninguém lembra de verdade dos próprios cliques. Ou pode deixar o programa contar.

O stack trace responde à pergunta *onde*. Os breadcrumbs respondem à outra metade, aquela que sempre te falta: **o que o usuário estava fazendo antes**. São migalhas que você deixa enquanto a aplicação trabalha, e o ExeWatch as anexa sozinho ao `Error` ou `Fatal` seguinte, em ordem, ao lado do stack.

Atenção, porém, a como se escrevem, porque é o ponto em que quase todo mundo erra da primeira vez. Não são cinco linhas uma embaixo da outra: cada uma fica onde a coisa acontece de verdade, espalhada pela aplicação, e entre uma e outra passam telas e minutos de trabalho do usuário.

```
// TFormFattura.FormShow -- o usuário abre uma fatura
EW.AddBreadcrumb(btNavigation, 'ui', 'Fatura aberta: ' + IntToStr(AInvoiceId));

// ... aqui o usuário trabalha: percorre as linhas, corrige uma base de cálculo, salva
...

// TFormUtente.CambiaRuolo -- o usuário muda de perfil
EW.AddBreadcrumb(btUser, 'auth', 'Mudou para o perfil admin');

// ... aqui o usuário ainda faz outras coisas, talvez por uns quinze minutos ...

// TImportListino.Esegui -- o usuário importa um arquivo externo
EW.AddBreadcrumb(btFile, 'io',
  Format('Importado %s (%d linhas)', [ExtractFileName(AFileName), ARowCount]));

// TFatturaDAO.RicaricaRighe -- acontece por baixo, o usuário nem vê
```

```
EW.AddBreadcrumb(btQuery, 'db', 'Linhas da fatura recarregadas');

// TFormFattura.BtnEsportaClick -- o último passo antes do crash
EW.AddBreadcrumb(btClick, 'ui', 'Clicou em Exportar');
EsportaFattura(AInvoiceId);   // <-- aqui estoura a exceção, e as cinco migalhas a
acompanham
```

Note também os valores dentro das mensagens: o número da fatura, o nome do arquivo, a contagem de linhas. Nas tags aqueles valores nunca entram, porque cada valor distinto vira uma dimensão e faz o seu dashboard explodir. Já em uma mensagem de breadcrumb eles ficam ótimos, e é justamente ali que se tornam úteis, porque a diferença entre "importou um arquivo" e "importou tabela_precos_2026.csv com 1284 linhas" é toda a distância entre um rastro e um diagnóstico.

Quando aquele crash chegar ao dashboard você não está mais lendo que existe uma access violation dentro de `SaveDocument`. Está lendo que aquele cliente, e só aquele cliente, importa uma tabela de preços externa antes de exportar, e que o seu código de export nunca viu linhas com aquele formato. A sequência que você não conseguia fazer com que te contassem está escrita ali.

A mesma coisa serve na versão mais escorregadia do problema, aquela em que o erro acontece em muitos clientes mas nem sempre. Com uns dez crashes no dashboard você para de raciocinar por hipóteses e começa a comparar os rastros: se em todos aparece uma certa passagem e nas sessões saudáveis ela nunca aparece, você terminou de adivinhar. É o mesmo trabalho que você faria com os logs, mas sem precisar ter previsto de antemão qual log iria te servir.

O tipo é um de um conjunto fixo de dezesseis valores (`btClick`, `btNavigation`, `btHttp`, `btQuery`, `btUser`, `btForm`, `btFile`, `btState`, `btTransaction`, `btConfig`, `btCustom` e outros), e serve ao dashboard para dar um ícone e tornar o rastro legível num relance. Existe também a forma curta, `EW.AddBreadcrumb('Exportação iniciada')`, quando te basta deixar uma anotação.

Quatro comportamentos para conhecer, porque mudam o que você encontra pela frente quando precisar:

- **O rastro é por thread, e as threads não se enxergam entre si.** Cada thread guarda as próprias migalhas. Se o usuário clica em Exportar na thread principal e o erro depois explode em uma thread de segundo plano, no rastro anexado àquele erro o clique não está. Quando você inicia um trabalho assíncrono, deixe uma migalha também dentro do worker com o que você passou para ele, senão a história se quebra exatamente no ponto em que você precisa dela.
- **Ficam as últimas 20 por thread.** A vigésima primeira derruba a mais antiga, portanto o que se anexa a um crash são os instantes imediatamente anteriores e não a sessão inteira. Por isso elas não vão dentro de um laço: vinte iterações de `btQuery` enchem o rastro e apagam o contexto que as precedia.

- **Só são anexadas a `Error` e `Fatal`.** Um `Info` ou um `Debug` não as leva junto, e sozinhas elas nunca são enviadas. Portanto não consomem cota até que uma falha as use.
- **São consumidas.** Uma vez anexado a um erro, o rastro daquela thread é esvaziado, assim o segundo erro não arrasta consigo os instantes anteriores do primeiro. Mas é preciso saber disso: se a mesma operação falhar duas vezes seguidas, o segundo evento carrega só as migalhas deixadas nesse meio-tempo. Diante de um par de erros, aquele com a história completa é o primeiro.

O que vale a pena deixar como breadcrumb: a passagem de uma tela ou de um diálogo para outro (`btNavigation`, `btForm`), as chamadas externas que você faz (`btHttp`), as queries que importam (`btQuery`), as ações com que o usuário muda o estado do programa, login, mudança de perfil, mudança de empresa (`btUser`), e os arquivos que ele abre ou importa (`btFile`). A regra prática é simples: se amanhã, diante de um crash, você teria vontade de perguntar "mas antes o que ele tinha feito?", aquela é uma migalha para deixar hoje.

O que o dashboard te devolve: você abre o crash e encontra, ao lado do stack, os últimos vinte passos que levaram até ali, em ordem, com tipo e categoria. O mesmo crash vindo de dois clientes diferentes se lê como duas histórias diferentes, e é aí que costuma saltar aos olhos qual das duas é a sua.

Receita #3: Você está prestes a brigar por uma funcionalidade que talvez ninguém use

> Uma reunião de planejamento. Alguém tem certeza de que a funcionalidade de export em lote é essencial e pede duas semanas para estendê-la. Outra pessoa acha que quase ninguém a toca. Os dois estão chutando, porque nenhum dos dois tem um número, e nesse tipo de discussão costuma vencer a voz mais alta. É exatamente o tipo de pergunta que um único counter encerra de uma vez por todas.

Coloque um counter no ponto de entrada de cada funcionalidade que te interessa. Aqui a ferramenta certa é um counter, não um log, porque a pergunta é "quantos", e os counters são somados em rollup no backend, assim o que você relê é o total em cada instalação, a baixo custo, sem que você guarde uma linha por clique.

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

TIP Incremente uma vez por uso lógico, não a cada iteração. Se gerar um relatório

> processa 500 linhas, continua sendo um único uso da funcionalidade, portanto um único incremento, não 500. Contando as linhas você mediria o tamanho dos relatórios, não quantas vezes as pessoas usam o export, e é a segunda a pergunta que você queria responder. Se a primeira também te interessa, essa é um gauge à parte: `EW.RecordGauge('report.rows', RowCount, 'reporting')`.

Um mês depois o dashboard ordena as suas funcionalidades por uso real, e quem encerra a discussão sobre o roadmap são os fatos: você estende o que as pessoas usam e retira em silêncio o que elas não usam.

Receita #4: Uma release "parece mais lenta" e ninguém consegue provar

> Você lança a v4.2. Em uma semana dois clientes comentam que o relatório principal "parece arrastado desde a atualização". É real, ou é a desconfiança de sempre que acompanha toda mudança? Você não consegue dizer, porque "parece mais lenta" não é um dado, e pedir ao cliente que cronometre no relógio não é um plano.

A solução é medir a operação em campo e etiquetar cada medida com a release que a produziu. Envolva a operação em um par de timings, e defina o rótulo de release no init para que cada amostra saiba a sua versão:

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

WARNING

Feche sempre o par em um `try..finally` (ou `try..except`). Se `RenderReport` levantar uma exceção, `EndTiming` precisa ser executado do mesmo jeito, senão o timing fica aberto e aquela amostra se perde, justamente nos caminhos de erro que você mais quer ver.

O rótulo de release vem do init: `InitializeExeWatch(ApiKey, CustomerId, '4.2.0')` define o `AppVersion` para a execução, enquanto a versão do binário é detectada à parte automaticamente, assim você obtém as duas. Eis agora o fluxo que responde à pergunta: depois que a release ficou em campo tempo suficiente para colher amostras, abra a página "Timing", filtre por janela temporal e

use "Export CSV". O arquivo respeita os filtros ativos e abre no Excel, onde você cria uma tabela dinâmica de duração média e p95 por `app_version`. A lentidão deixa de ser uma opinião e vira "report.render passou de 120ms para 300ms na 4.2", coisa que você captou nos seus próprios dados de campo antes que virasse cancelamentos. Uma visão nativa "filtrar por versão" dentro da interface está no backlog; enquanto ela não chega, a tabela dinâmica sobre o CSV é o caminho suportado, e o mesmo export vale do mesmo jeito para "Logs", "Timing" e "Metrics".

Receita #5: "O checkout está lento", mas lento onde?

> Os clientes dizem que o checkout demora uma eternidade. Você cronometra tudo e dá quatro segundos. Aquele número sozinho é quase inútil, porque quatro segundos de quê? Carregar o carrinho? O gateway de pagamento? Renderizar o recibo? Otimizando às cegas você poderia passar um dia inteiro deixando a query de banco mais rápida e mover o total de quatro segundos para três e nove, porque o custo de verdade estava todo em outro lugar.

Os traces aninhados quebram aqueles quatro segundos. Inicie um trace, faça um timing rodar em volta de cada fase lá dentro, e feche o trace. O backend reconstrói uma cascata, uma barra por fase, assim você vê qual filho de fato domina:

```
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` retorna um id de trace e `EndTrace` retorna os milissegundos totais decorridos, assim você pode registrar ou fazer assert sobre o total se quiser, e cada timing aninhado vira um span filho. No dashboard "o checkout está lento" vira "o pagamento é 80% do checkout", o que te diz que o

problema é o gateway, não o seu código, e te poupa o dia que você teria passado otimizando a fase errada. .NET e Python espelham tudo isso ao pé da letra; a DLL usa `ew_StartTrace(Name, Buffer, BufLen)`, que escreve o id de trace em um buffer fornecido por quem chama, e `ew_EndTrace(&ElapsedMs)`.

Um trace agregado: cada fase é uma barra com o seu avg, min, max e p95, e a porcentagem te diz quanto do pai ela representa. Aqui as fases de transform e render dominam, então é ali que o tempo vai embora. As estatísticas são calculadas sobre as execuções bem-sucedidas, assim uma fase que falhou não as distorce.

O que acontece se você esquecer de fechar um timing

Mais cedo ou mais tarde você vai esquecer um `EndTiming`. O SDK é construído para sobreviver a isso sem vazar memória nem reportar um número errado, mas o que você obtém nunca é tão bom quanto um par limpo, e essa é a verdadeira razão do try/finally. Eis exatamente o que acontece com um timing aberto, dependendo de como ele termina:

- **Nada, se ele simplesmente ficar aberto.** Um timing sem um `EndTiming` correspondente nunca emite uma amostra. Não é contado nem entra na média, simplesmente não existe. Você perde em silêncio aquela medida específica.
- **Fechado automaticamente como falho se você reiniciar o mesmo id.** Chame `StartTiming('report.render')` enquanto um `report.render` anterior ainda está aberto na mesma thread e o SDK fecha o antigo para você, marcado `success = false` e sinalizado `auto_closed`, com um Warning nos seus logs ("auto-closed, duplicate StartTiming"). Por ter falhado, ele fica fora das estatísticas de duração; o Warning está ali para te dizer que o código tem um furo.
- **O mais antigo é descartado se muitos se acumularem.** Cada thread guarda no máximo 100 timings abertos. Inicie o 101º e o aberto mais antigo é fechado à força como falho, de novo com um Warning. É a rede de segurança que impede que um lento vazamento de timings esquecidos cresça sem limite.
- **Um trace limpa os próprios filhos.** Se o timing esquecido está aninhado dentro de um `StartTrace`/`EndTrace`, `EndTrace` fecha à força qualquer span filho que você tenha deixado aberto,

marcado como falho, assim a cascata continua completa.

O padrão nos quatro casos: um `EndTiming` esquecido vira uma amostra falha mais um Warning, nunca uma duração limpa. É o mecanismo que te protege, não uma funcionalidade em que você deva se apoiar. Envolva cada par em um try/finally e nada disso jamais dispara.

Receita #6: O app vai bem às 9h e arrasta às 17h

> Um cliente relata que a sua aplicação está ágil de manhã e arrastada no fim do dia, e que um reinício resolve até o dia seguinte. Aquele comportamento é um vazamento de recursos, e é invisível para um crash reporter, porque nada dá crash. Só degrada, em silêncio, ao longo de horas, em uma máquina que você nunca vê.

Um vazamento é um nível que sobe e não volta mais a descer, que é exatamente o que um gauge mede. Você não quer espalhar chamadas de gauge por todo o seu laço de render; você quer o valor amostrado em intervalos. Registre um gauge periódico e a thread de amostragem do SDK lê o seu callback por você, sem timer nenhum da sua parte:

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

Coisas boas de observar assim: working set em MB, número de GDI ou de handles, número de documentos abertos, tamanho da cache. Depois de um ou dois dias o dashboard mostra a assinatura com clareza, um dente de serra que sobe pela sessão inteira e despenca no reinício, e como um gauge guarda média, min, max e número de amostras por janela, a média e o max em crescimento são o vazamento. Fatie por `app_version` e muitas vezes você consegue cravá-lo na release exata que o introduziu.

Uma divergência a respeitar: o callback do gauge periódico existe em Delphi, .NET (um `Func<double>`), Python e JS, mas não na DLL, porque um callback não consegue atravessar a ABI C flat. Com a DLL o intervalo pertence ao host: faça rodar um timer seu e chame `ew_RecordGauge(Name, Value, Tag)` a cada tick.

Receita #7: É todo mundo, ou um cliente só?

> Os seus logs mostram uma rajada de erros de sync e o primeiro instinto é que o parque de máquinas inteiro está falhando. Antes de entrar em pânico, pergunte-se se está atingindo todo mundo ou um cliente só com uma configuração incomum, e note que percorrer logs crus de 200 instalações nunca vai te dizer isso.

Se você define o id do cliente no init e etiqueta os logs por subsistema, consegue isolar de forma limpa qualquer fatia:

```
EW.SetCustomerId('acme-corp');
// ... mais adiante, no caminho de sincronização:
EW.Error('Sincronização rejeitada pelo servidor', 'sync');
```

Agora filtre o dashboard nos erros `sync` da Acme e o quadro se esclarece em poucos segundos: é um cliente só, atrás de um firewall corporativo, não o seu código falhando em todo lugar. Dê mais um passo e crie um alerta com escopo `customer_external_id` e tag, e você recebe a notificação específica sobre os problemas de sync da Acme, ficando em silêncio sobre os outros 200 clientes que estão bem. Taxas de erro por cliente e por funcionalidade, cada uma com o seu alerta, tudo fruto de uma etiquetagem coerente.

Receita #8: Com que frequência ele falha de verdade?

> "O sync às vezes falha" é o tipo de relato que esconde a única coisa que você precisa saber: quão frequente é "às vezes"? Uma vez por semana é um dar de ombros; uma tentativa em três é um incidente. Você não consegue priorizar enquanto não vir a proporção.

A versão mais simples é um counter por resultado, fatiado com uma tag de resultado:

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

Mas se você já está cronometrando a operação, não precisa de um segundo counter, porque `EndTiming` leva uma flag `success` como terceiro parâmetro (padrão `True`):

```
EW.StartTiming('sync', 'sync');
try
  DoSync;
  EW.EndTiming('sync', nil, True);   // deu certo
except
  EW.EndTiming('sync', nil, False);  // falhou, mas o timing fecha do mesmo jeito
  raise;
end;
```

Essa flag faz algo mais sutil que um counter, e vale a pena entender. Um timing falho não é descartado: ele é registrado, assim você pode contar as falhas e vê-las na lista dos timings. Mas fica de fora das estatísticas de duração. Média, min, max e p95 na página "Timing" são calculados só a partir das execuções bem-sucedidas. Isso importa, porque as falhas costumam ser as chamadas mais lentas, uma operação que desistiu depois de um timeout de 30 segundos, e se aquelas contassem na sua latência você pensaria que o sync ficou mais lento quando na verdade ele começou a falhar. Com a flag success você lê a verdadeira latência das execuções bem-sucedidas e a taxa de falha, a partir de uma única chamada.

Counter, gauge ou timing: qual

Estas três ferramentas são confundidas o tempo todo, e confundi-las não lança um erro, registra em silêncio dados que não significam nada, o que é pior. A distinção é simples, uma vez dita em voz alta:

Ferramenta	Responde a	Como agrega	O que te custa errar
Counter	quantos, com que frequência	somado na janela	um gauge para "número de exports" joga fora o total
Gauge	qual é o valor neste momento	média, min, max, número de amostras por janela	um counter para "profundidade de fila" soma níveis e vira um absurdo
Timing	quanto tempo demorou	distribuição de duração, com trace	um counter para "quão lento" te dá uma contagem, não uma duração

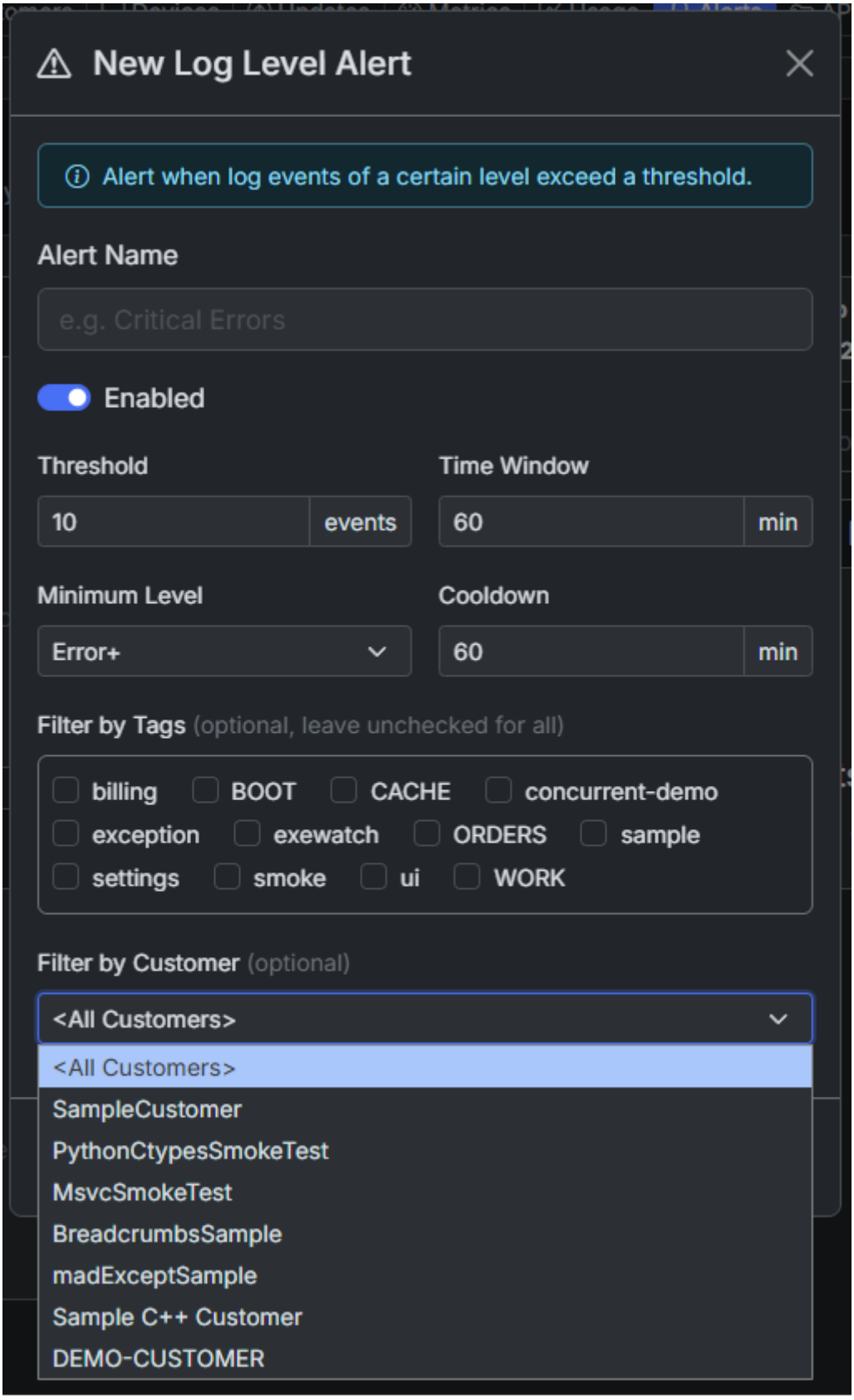
Na dúvida, tente somar duas leituras. Se a soma faz sentido, 12 exports mais 8 exports dão 20 exports, é um counter. Se não faz sentido, uma fila de 12 mais uma fila de 8 não dá uma fila de 20, é

um gauge. Se o que te interessa é o tempo decorrido, é um timing.

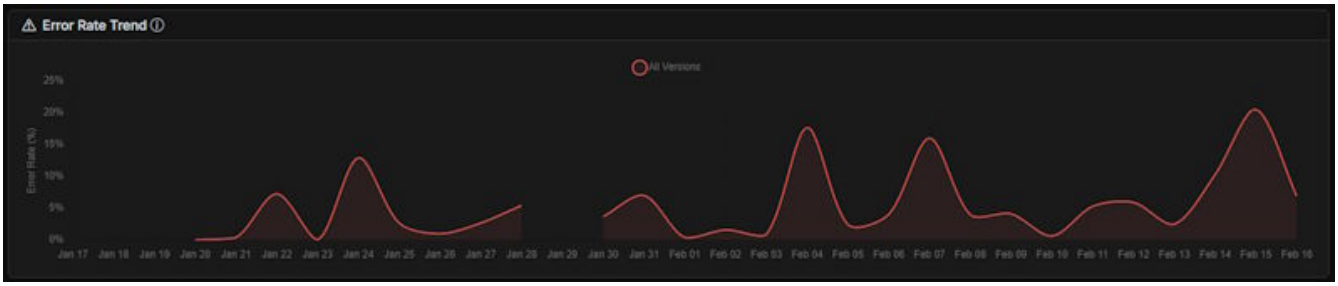
Alertas que importam

Um alerta é o que te permite parar de olhar o dashboard. Em vez de conferir de vez em quando torcendo para flagrar um problema, você descreve o problema uma vez e o ExeWatch te manda um e-mail quando ele acontece. Hoje existem dois tipos, e convém saber com precisão sobre o que cada um dispara, porque os limites honestos do alerting moldam o modo como você usa tudo o que vem acima.

Os alertas de volume de log disparam quando a contagem dos eventos de log iguais ou superiores a um nível ultrapassa um limiar dentro de uma janela. Você define `threshold` (uma contagem), `window_minutes`, `min_level` (padrão `error`), um filtro `tags` opcional, um `customer_external_id` opcional e `cooldown_minutes` (padrão 60) para que um único incidente não te notifique cinquenta vezes. Um típico diz: "mais de 20 eventos `fatal` em 15 minutos para o cliente Acme".



O mesmo alerta como formulário no console, em "Configure alerts": um limiar sobre uma janela temporal em um nível mínimo, com um cooldown, se necessário restrito por tag e por cliente. Colocar "Filter by Customer" em Acme é o jeito de ser avisado dos erros da Acme sem ouvir falar de mais ninguém.



Taxa de erro ao longo do tempo. Um pico como esses aqui é exatamente o que um alerta de volume de log vigia, assim ele te alcança por e-mail no instante em que começa, em vez de quando você por acaso abrir este gráfico.

Os alertas de timing disparam quando as operações que correspondem a um padrão de id de timing rodam mais lentas que um limiar de duração, com frequência suficiente para ultrapassar um limiar de contagem dentro de uma janela. Os campos são um `timing_id_pattern`, um limiar de duração em milissegundos, um limiar de ocorrências `threshold` (padrão 5), `window_minutes`, um `customer_external_id` opcional e `cooldown_minutes` (padrão 240). É assim que você garante ser avisado de que "report.render está rodando lento em campo" sem ter que ficar vigiando.

NOTE

É igualmente importante saber o que os alertas hoje não fazem. Não existe um "alerta quando um gauge ultrapassa X" nem um "alerta quando um counter ultrapassa X". Os alertas disparam sobre contagens de eventos de log e sobre timing, não sobre valores arbitrários de métrica. Portanto se você quer ser notificado quando a memória ou a profundidade de fila ultrapassar um limiar, a resposta honesta é que ainda não dá; você olha o gauge no dashboard. O alerting por limiar sobre valores de métrica no momento não é uma funcionalidade.

O jeito de manter os alertas úteis em vez de ignorados: mantenha honestos os seus níveis de log (veja anti-pattern, senão o seu limiar `error` dispara em coisas que erros não são), dê a cada alerta um escopo por tag ou por cliente para que ele tenha um dono claro, e coloque um cooldown longo o bastante para que um incidente de verdade chegue como um único e-mail.

Comparar as versões com o export CSV

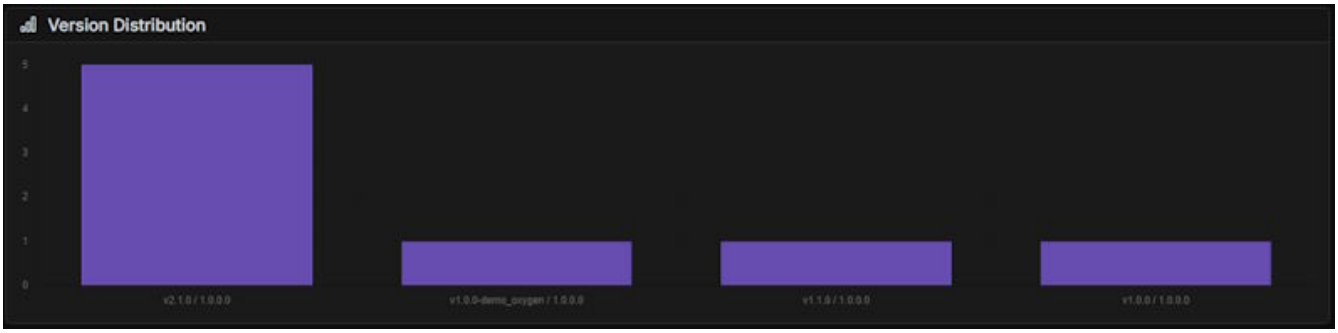
Você lança uma versão e quer saber se ela ficou mais lenta ou mais barulhenta que a anterior. Hoje essa comparação passa pelo export CSV, e vale a pena explicitar o fluxo porque ele responde a uma pergunta que os clientes fazem o tempo todo.

1. Lance a versão com o `AppVersion` definido no init (`InitializeExeWatch(ApiKey, CustomerId, '4.2.0')`). Cada evento agora sabe qual versão o produziu.
2. Dê tempo em campo para colher amostras, depois abra a página "Timing" e configure os filtros:

janela temporal, tag, cliente.

3. Use "Export CSV". O arquivo respeita aqueles filtros ativos e abre no Excel.
4. Crie uma tabela dinâmica de duração média e p95 por `app_version`, e a regressão, se houver, está ali na tabela dinâmica.

O mesmo export existe nas páginas "Logs" e "Metrics", assim você pode comparar o volume de erros ou os níveis de métrica entre as versões do mesmo jeito. Um filtro de timing por versão dentro da interface está no backlog; quando chegar, isto vai virar um par de cliques em vez de uma tabela dinâmica. Até lá, o caminho do CSV é o modo suportado para a comparação versão a versão, e funciona bem o bastante para que vários times parem por aí.



Como cada evento carrega o seu `AppVersion`, o console já sabe como as suas releases estão distribuídas em campo. É sobre esse mesmo rótulo de versão que você cria a tabela dinâmica do timing exportado.

Juntando tudo: instrumentar um método real

As chamadas isoladas são fáceis de aprovar com um aceno de cabeça e difíceis de posicionar. Eis então a instrumentação aplicada a código comum: um método de sync em um data module, do tipo que todo sistema de gestão em Delphi tem. Primeiro a versão nua, que faz o seu trabalho sem telemetria:

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

  FConnection.StartTransaction;
```

```
  try
    ImportInvoices(Response.ContentAsString);
    FConnection.Commit;
  except
    FConnection.Rollback;
    raise;
  end;
  FLastSync := NowUtcIso;
end;
```

Agora o mesmo método instrumentado, cada acréscimo faz exatamente um trabalho:

```
procedure TSyncModule.SyncInvoices;
var
  Response: IHTTPResponse;
begin
  EW.AddBreadcrumb(btHttp, 'sync', 'GET /invoices since ' + FLastSync);  // rastro
para um eventual crash
  EW.StartTiming('sync.invoices', 'sync');                               // cronometra
a operação inteira
  try
    Response := FHttp.Get(FBaseUrl + '/invoices?since=' + FLastSync);
    if Response.StatusCode <> 200 then
      raise ESyncError.CreateFmt('Sincronização rejeitada pelo servidor: HTTP %d',
[Response.StatusCode]);

    EW.StartTiming('sync.import', 'sync');                               // isola a
fase de banco de dados
    FConnection.StartTransaction;
    try
      ImportInvoices(Response.ContentAsString);
      FConnection.Commit;
      EW.EndTiming('sync.import', nil, True);
    except
      FConnection.Rollback;
      EW.EndTiming('sync.import', nil, False);                           // execução
falha, fora das estatísticas
      raise;
    end;

    FLastSync := NowUtcIso;
    EW.IncrementCounter('sync.completed', 1, 'sync');                    // um para
cada sync bem-sucedido
    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;
```

O que foi parar onde, e por quê:

- O **breadcrumb** fica no topo, antes da chamada que descreve, para que, se algo mais abaixo levantar uma exceção, o rastro já registre a requisição que levou até lá.
- O **timing externo** (`sync.invoices`) envolve a operação inteira; o **timing interno** (`sync.import`) isola a fase de banco de dados, assim a cascata separa o tempo de rede do tempo de import.
- O **servidor que responde mal levanta uma exceção**, como qualquer outra falha: com um status diferente de 200 não há nada para importar, e quem chamou precisa saber que a sincronização não aconteceu. Note que isso não acrescenta instrumentação, tira: a falha HTTP acaba no mesmo handler lá embaixo, que a registra com a sua classe de exceção e os breadcrumbs, em vez de ter um ramo que precisa escrever os seus próprios logs e fechamentos de timing.
- Cada `EndTiming` está tanto no caminho de sucesso quanto no de falha, com `False` quando a operação falha, assim um sync quebrado não polui os seus números de latência.
- O **counter** incrementa uma vez, só em caso de sucesso, assim `sync.completed` é uma contagem verdadeira dos syncs bem-sucedidos.
- O **erro** é registrado com a exceção no handler mais externo, onde carrega o stack e os breadcrumbs aqui em cima, e depois é relançado para que o tratamento de erros do app fique inalterado.

Note a forma: a instrumentação emoldura o código real, não o substitui. Apague cada linha `EW.` e o método continua funcionando exatamente como antes. A instrumentação se limita a observar.

Anti-pattern

Nenhum destes é "você registrou demais". São os casos em que o que você manda te engana em vez de te ajudar, e a correção está ao lado de cada um.

Repetir a mesma linha dentro de um laço. Uma chamada `Debug` em um laço quente, ou um `IncrementCounter` a cada iteração. A questão não é a quantidade: é que quinhentas linhas idênticas respondem todas à mesma pergunta que a primeira já tinha respondido, e nesse meio-tempo consomem a cota que servia aos eventos de verdade. Registre a operação, não as suas iterações: uma linha quando ela começa, uma quando termina ou falha, e um único incremento de counter

por operação lógica. Se o detalhe por iteração te serve mesmo durante um diagnóstico, mantenha-o em nível `Debug` e suba o nível mínimo da aplicação pelo console quando terminar.

Dados pessoais nos logs. E-mails, nomes, tokens ou conteúdos de arquivos em uma mensagem ou em uma tag. Os logs são conservados e exportados em CSV, e as tags são dimensões de baixa cardinalidade, não payload, então isso é ao mesmo tempo um problema de privacidade e uma bagunça. Registre ids e categorias; use o campo id do cliente para a identidade.

Níveis que mentem. Tudo registrado em `Error`, ou falhas de verdade registradas em `Info`. Os alertas usam por padrão `min_level = error`, portanto se os seus níveis estão errados os seus alertas também estão, e ou você é notificado à toa ou nunca é notificado. A escala que os mantém honestos: Fatal significa que o app não pode continuar, Error que uma operação falhou, Warning degradado mas funcionando, Info uma mudança de estado notável, Debug detalhe só para desenvolvedores.

Gauge e counter, trocados. Um counter para "profundidade de fila atual" soma níveis em um total sem sentido; um gauge para "número de exports" joga fora o total. Aplique o teste da adição da tabela aqui em cima antes de escolher.

Tags que explodem. Um id, um timestamp, ou qualquer valor por requisição em uma tag. As tags são dimensões com um pequeno conjunto fixo de valores; valores ilimitados se multiplicam em milhares de dimensões descartáveis e tornam o dashboard inutilizável. Mantenha o vocabulário pequeno.

Registrar de novo o que você já ganha de graça. Registrar na mão OS, versão ou hardware que já viajam no snapshot do dispositivo a cada lote. Não faça isso. O único campo que você define na mão é o `AppVersion`, o rótulo de release.

O que acontece se a conexão com a internet cair?

Os apps desktop rodam em notebooks que entram em túneis, em máquinas atrás de VPNs caprichosas, em um PC que alguém desconecta da rede ao sair. Então a pergunta é legítima: o que acontece com a sua telemetria quando o SDK não consegue alcançar o servidor?

Não se perde nada. O SDK não envia os eventos direto do ponto em que você os chama. Ele os acumula em um buffer, escreve em disco, e quem cuida de enviá-los é uma thread de envio em segundo plano. Quando a rede está fora o envio simplesmente falha, os arquivos ficam em disco, e a thread tenta de novo em intervalos. No instante em que a conectividade volta, os arquivos na fila são enviados em ordem. Um crash registrado no avião aparece no seu dashboard na primeira vez que o notebook volta a ficar online.

Quem governa isso são quatro configurações do `TExeWatchConfig`:

- `StoragePath` é onde vivem os arquivos em espera. Por padrão é uma pasta por app; aponte-o para um lugar em que o seu app sempre consiga escrever.
- `FlushIntervalMs` (padrão 5000) é de quanto em quanto o buffer em memória é escrito em disco. Mais curto significa menos dados em risco se o processo for morto no meio do caminho; mais longo significa escritas menos frequentes e maiores.
- `RetryIntervalMs` (padrão 30000) é de quanto em quanto a thread de envio tenta de novo depois de uma falha. É a sua cadência de reconexão: um valor mais baixo escoa o acúmulo mais rápido assim que a rede volta, ao custo de mais tentativas enquanto ela ainda está fora.
- `MaxPendingAgeDays` (padrão 7) limita por quanto tempo os arquivos não enviados são guardados. Uma máquina offline por mais tempo que isso descarta os dados mais antigos em vez de enviar na reconexão uma onda de semanas atrás. Coloque em 0 para guardar tudo, sem limite.

O acúmulo você mesmo pode acompanhar: `GetPendingCount` retorna quantos eventos ainda estão esperando para ser enviados.

NOTE

Uma conexão caída não é um `429`, ainda que os dois terminem com o dado não chegando ao servidor. São opostos. Uma falha de rede é temporária, então o SDK guarda o dado e tenta de novo. Um `429` quer dizer que você ultrapassou a cota, o que é intencional, então o SDK descarta o dado em vez de tentar de novo. Perder telemetria por uma conexão instável exige um apagão mais longo que `MaxPendingAgeDays`; perdê-la por um `429` exige apenas estourar o seu plano, e é disso que trata a próxima seção.

Instrumente o que tem significado, no nível certo

Um fato técnico para conhecer, e ele chega no fim do guia de propósito: a cota mensal conta os eventos que você nos manda, não os que ficam armazenados. Contam juntos os logs e as atualizações das métricas, ou seja, counters, gauges e timings, não só os logs. As mensagens internas do SDK, aquelas com tag `ew.system`, não entram na contagem. E como conta o que chegou, apagar dados libera espaço mas não devolve cota.

Não é um convite a instrumentar menos. A ordem certa é a que você seguiu até aqui: primeiro decida o que quer ver, depois, se e quando o volume virar um tema, você o governa. As ferramentas para governá-lo existem e ninguém está te pedindo para abrir mão de um dado que te interessa.

- O nível mínimo se define por aplicação, no console, e o SDK o recebe na conexão seguinte. Você

pode desenvolver com todo o `Debug` que quiser e depois manter em produção só `Info` para cima, sem tocar no código e sem lançar uma build nova.

- Também pelo console, a amostragem manda só uma fração dos eventos de rotina. `Error` e `Fatal` sempre passam por cima dela, portanto você pode ralear um log verboso em dez mil instalações sem perder um único crash.
- Agregue o que é pura repetição: um incremento por operação lógica em vez de um por iteração de laço.
- Use as tags para fatiar, assim uma métrica bem etiquetada te poupa dez métricas quase idênticas.

Se você está sempre perto do teto com uma instrumentação que te serve toda, não há nada a cortar: você está acompanhando mais aplicações ou mais clientes do que quando escolheu o plano, e a essa altura convém passar para o nível seguinte.

NOTE

Ultrapassada a cota mensal o SDK recebe um `429` e começa a descartar os dados, então convém ficar de olho no uso pelo console. Não para registrar menos, mas para perceber a tempo que o plano ficou apertado, em vez de descobrir isso perdendo os eventos de um pico.

No roadmap

Algumas coisas que as pessoas pedem estão planejadas mas ainda não lançadas. Estão listadas aqui, honestamente, para que você saiba onde estão hoje as fronteiras e não vá procurar uma funcionalidade que ainda não existe.

- **Detectar quando um app fica em silêncio.** Um dead-man’s-switch, para que você fique sabendo do cliente cujo app parou de dar sinal de vida, não só daquele cujo app levantou um erro. Hoje os alertas disparam sobre eventos que acontecem, não sobre eventos que param de acontecer, portanto isso ainda não é possível. Planejado.
- **Detecção de anomalias.** Fazer emergir automaticamente um "aqui tem algo estranho" nas métricas, em vez de você configurar limiares fixos. Ainda não implementado.
- **Uma página de usage analytics.** Uma visão dedicada à adoção e aos padrões de uso das funcionalidades, além dos counters crus. Ainda não implementada.
- **Uma API de leitura / consulta.** Hoje as API keys são apenas de ingest, portanto tirar os seus dados para relatórios automatizados espera por ela. O export CSV é o caminho manual nesse meio-tempo.

Para tudo o que existe hoje, a referência do SDK no console tem as assinaturas exatas, as chaves de configuração e os passos de instalação. Este guia é o quê e o porquê; aquela é o como.