THOMAS MORE

Lab 4.3

Tracing Requests with Correlation IDs and Operation IDs

Toepasbare Cloud Security voor IT-Professionals

2025 - 2026

Contents

1. Tracing Requests with Correlation IDs and Operation IDs

1.1. Context

A single user request to your application may pass through multiple services: frontend, public backend, internal backend, Key Vault, and Azure Table Storage. Without identifiers that link these logs together, each service logs independently and you cannot connect the dots during an investigation.

This lab has two parts: first, you verify that the OpenTelemetry-based SDK (`@azure/monitor-opentelemetry`) is properly instrumented in your backend code so that calls to Key Vault and Table Storage are automatically captured as telemetry, then you query those traces in Log Analytics to follow a request through the full stack.

1.2. Learning goals

By the end of this lab, you should be able to explain:

- How correlation IDs (the `x-correlation-id` header) are generated, propagated, and used across services.
- The difference between a correlation ID (set by the client in the `x-correlation-id` header) and an `operation_Id` (generated automatically by the SDK for every HTTP request).
- How the OpenTelemetry-based SDK (`@azure/monitor-opentelemetry`) automatically captures Azure SDK calls as dependencies.
- How to query the `AppDependencies` table in Log Analytics to trace calls from your app to Azure services.
- How to join requests → dependencies → traces using `operation_Id` for end-to-end tracing, and how correlation IDs let you follow a specific client request across services.

> **Note**
>
> Two identifiers, two purposes: the **correlation ID** (from the `x-correlation-id` header) lets you trace a specific client request across services. The **`operation_Id`** is generated automatically by the SDK for every HTTP request and links all telemetry records (AppRequests, AppTraces, AppDependencies) from that request together.

1.3. Estimated time

65 minutes

Suggested timing:

Time	Activity
10 minutes	Verify SDK instrumentation in the backend code
10 minutes	Generate traffic with custom correlation IDs
15 minutes	Query dependencies and traces in Log Analytics
15 minutes	Trace a request from public to internal backend

10 minutes	Trace the full data flow through Key Vault
5 minutes	Reflection and discussion

1.4. Before you start

Make sure you have:

- [ ] Day 4 monitoring infrastructure is enabled (pipeline ran — Lab 4.1)
- [ ] Diagnostic settings are configured (Lab 4.2)
- [ ] You know the URL of your public backend App Service
- [ ] You have access to the Azure portal and the Log Analytics workspace

2. Verify SDK Instrumentation

The OpenTelemetry-based SDK (`@azure/monitor-opentelemetry`) has already been added to both backends. Your task is to verify it is correctly instrumented.

2.1. Check the SDK dependency

Open `apps/backend/package.json` and confirm that `@azure/monitor-opentelemetry` is listed in the dependencies:

```
"@azure/monitor-opentelemetry": "^1.x.x"
```

Also check `apps/backend-internal/package.json` — it includes both `@azure/monitor-opentelemetry` and `@azure/opentelemetry-instrumentation-azure-sdk`.

2.2. Verify SDK initialization

Open `apps/backend/src/index.ts`. You will see the SDK initialized at the top of the file:

```
import { useAzureMonitor } from "@azure/monitor-opentelemetry";

const connectionString = process.env.APPLICATIONINSIGHTS_CONNECTION_STRING;

if (!connectionString) {
  console.warn("APPLICATIONINSIGHTS_CONNECTION_STRING is not set. Telemetry will not
be exported.");
} else {
  useAzureMonitor({
    azureMonitorExporterOptions: {
      connectionString,
    },
  });
}
```

> **Question**
>
> Why does the SDK use an environment variable (`APPLICATIONINSIGHTS_CONNECTION_STRING`) instead of hard-coding the connection string? What are the security and operational implications?

2.3. Verify the internal backend is also instrumented

Open `apps/backend-internal/src/index.ts`. The same `useAzureMonitor()` call is present there too. This means Table Storage calls from the internal backend will be captured as dependencies — enabling complete end-to-end tracing.

2.4. Verify correlation middleware

Open `apps/backend/src/correlation.ts`. You will find the `correlationMiddleware` function that reads the `x-correlation-id` header from incoming requests and generates a UUID if it is missing. This middleware is wired into Express at the top of `app.ts`:

```
app.use(correlationMiddleware);
```

The same middleware is used in the internal backend. Every request — whether it carries a correlation ID or not — will have one attached, which ensures all log entries can be traced back to the original client request.

2.5. Verify structured logging

Open `apps/backend/src/logger.ts`. You will find a `log()` function that emits structured telemetry via the OpenTelemetry logs API (`@opentelemetry/api-logs`). Throughout the application code, every endpoint calls `log()` with an event name and optional fields:

```
log("public-api-request-received", { correlationId, route: "/api/data-demo" });
```

These structured log entries appear in the `AppTraces` table in Log Analytics, where you can query them by correlation ID or event name.

3. Generate Traffic with Custom Correlation IDs

Run these requests, each with a custom correlation ID:

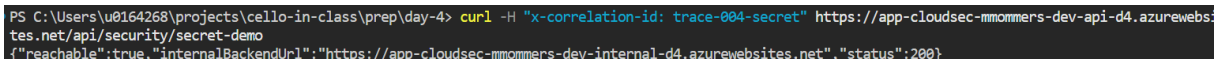
```
# Request 1: Health check
curl -H "x-correlation-id: 01111111-1111-4111-b111-111111111111" https://<your-backend-url>/api/health

# Request 2: Public API
curl -H "x-correlation-id: 01111111-1111-4111-b111-111111111112" https://<your-backend-url>/api/public-message

# Request 3: Trace demo (full stack trace)
curl -H "x-correlation-id: 01111111-1111-4111-b111-111111111113" https://<your-backend-url>/api/trace-demo

# Request 4: Key Vault access
curl -H "x-correlation-id: 01111111-1111-4111-b111-111111111114" https://<your-backend-url>/api/security/secret-demo

# Request 5: Data demo
curl -H "x-correlation-id: 01111111-1111-4111-b111-111111111115" https://<your-backend-url>/api/data-demo
```



3.1. Generate Suspicious Traffic

Run these requests to simulate attack patterns:

```
# Attack 1: Direct internal backend access
curl https://<your-internal-backend-url>/internal/data/welcome

# Attack 2: Multiple correlation IDs (scanning pattern)
for ($i = 1; $i -le 5; $i++) {
  curl -H "x-correlation-id: 01111111-1111-4111-b111-11111111$i" "https://<your-backend-url>/api/security/secret-demo"
}

# Attack 3: Rapid-fire enumeration
$endpoints = @(
  "/api/health",
  "/api/config",
  "/api/security/secret-demo",
  "/api/public-message"
)

foreach ($endpoint in $endpoints) {
  curl -s -o $null -w "%{http_code} $endpoint`n" "https://<your-backend-url>$endpoint"
}
```

Note

The suspicious traffic simulates patterns that a security analyst would investigate. The direct internal backend access bypasses network controls (but would be blocked). The multiple correlation IDs simulate a scanning tool. The rapid-fire enumeration simulates endpoint discovery.

Question

Why do the suspicious requests not always include an `x-correlation-id` header? What does this tell you about how attackers operate?

3.2. Wait for Data to Flow into Log Analytics

Note

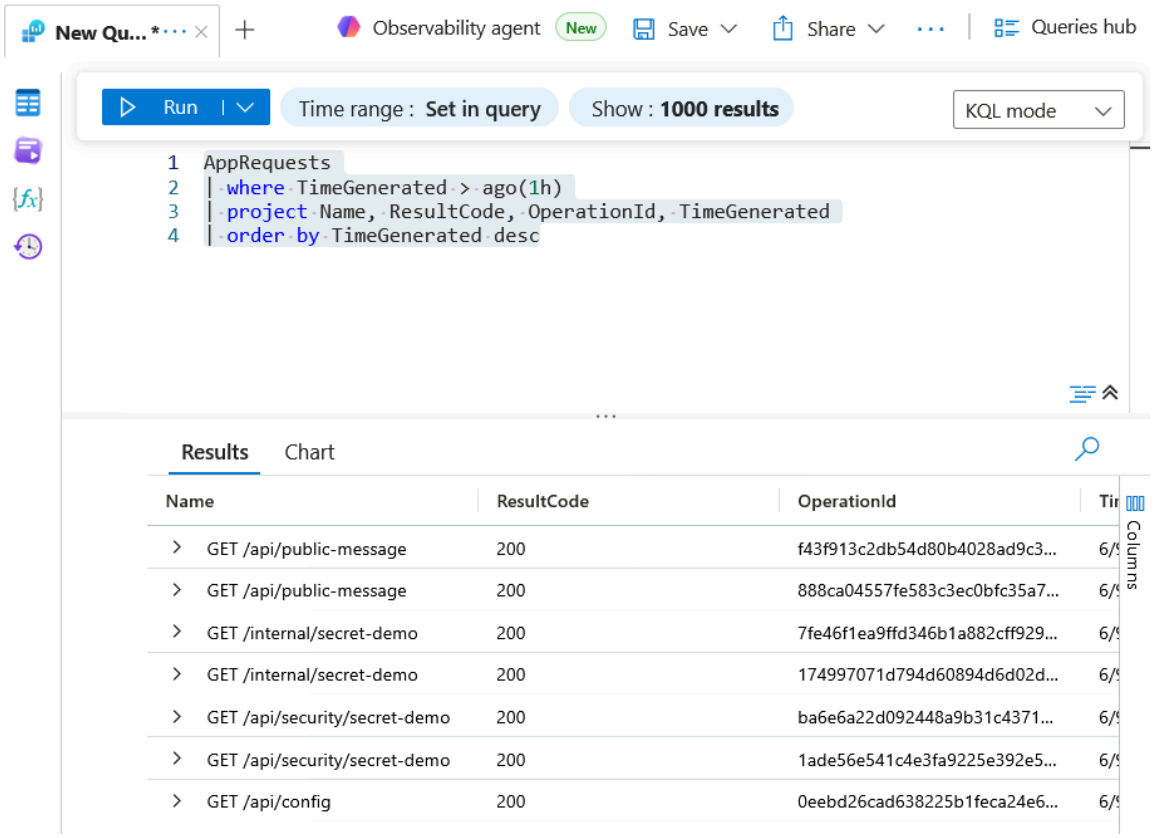
Log Analytics has a small ingestion delay — typically 30–60 seconds after traffic is generated. Wait at least one minute before running your queries, and if results are empty, try again after waiting a bit longer. If data still doesn't appear after several minutes, check that the SDK connection string is configured correctly on the App Service.

4. Query Application Insights for Correlation IDs

4.1. Find Requests by Correlation ID

In the Log Analytics **Logs** editor, run:

```
AppRequests
| where TimeGenerated > ago(1h)
| project Name, ResultCode, OperationId, TimeGenerated
| order by TimeGenerated desc
```



4.2. Find Traces by Correlation ID

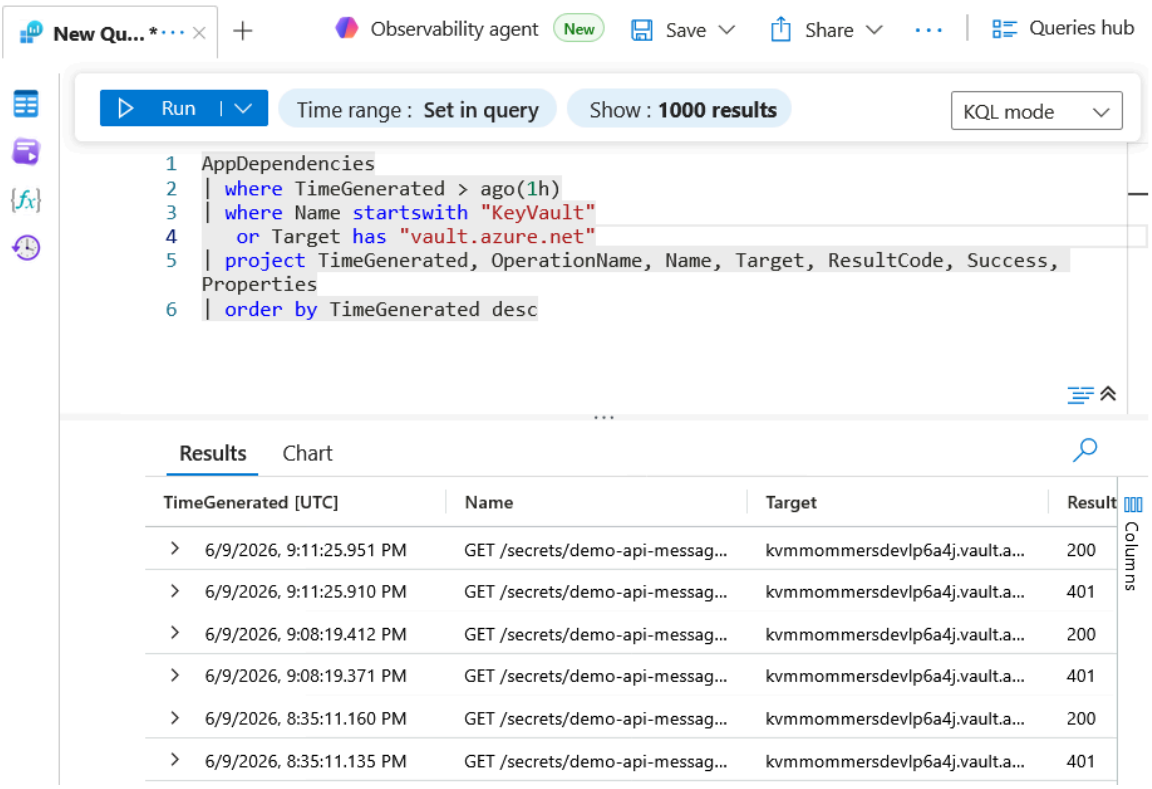
```
AppTraces
| where TimeGenerated > ago(1h)
| where Properties has "01111111-1111-4111-b111-111111111113"
   or Properties has "01111111-1111-4111-b111-111111111114"
| project TimeGenerated, Message
| order by TimeGenerated desc
```

Your answer: For the `01111111-1111-4111-b111-111111111113` correlation ID, list all log entries across the `AppRequests` and `AppTraces` tables. How many services logged this correlation ID?

4.3. Find Dependencies by Name (Key Vault Calls)

This is the key query — it shows Key Vault calls captured by the SDK:

```
AppDependencies
| where TimeGenerated > ago(1h)
| where Name startswith "KeyVault"
   or Target has "vault.azure.net"
| project TimeGenerated, OperationName, Name, Target, ResultCode, Success,
Properties, OperationId
| order by TimeGenerated desc
```



Question

Look at the dependencies table. Can you see entries for Key Vault calls? Do they have an `OperationId` that matches a request in the requests table? What does this tell you about how the SDK captures Azure SDK calls?

4.4. Find Dependencies by Name (Table Storage Calls)

```
AppDependencies
| where TimeGenerated > ago(1h)
| where Name startswith "TableStorage"
   or Target has ".tables.core.windows.net"
| project TimeGenerated, OperationName, Name, Target, ResultCode, Success,
Properties, OperationId
| order by TimeGenerated desc
```

Question

Do you see Table Storage dependency calls in the results? If so, what `OperationId` do they share with which request? What does this tell you about end-to-end tracing?,

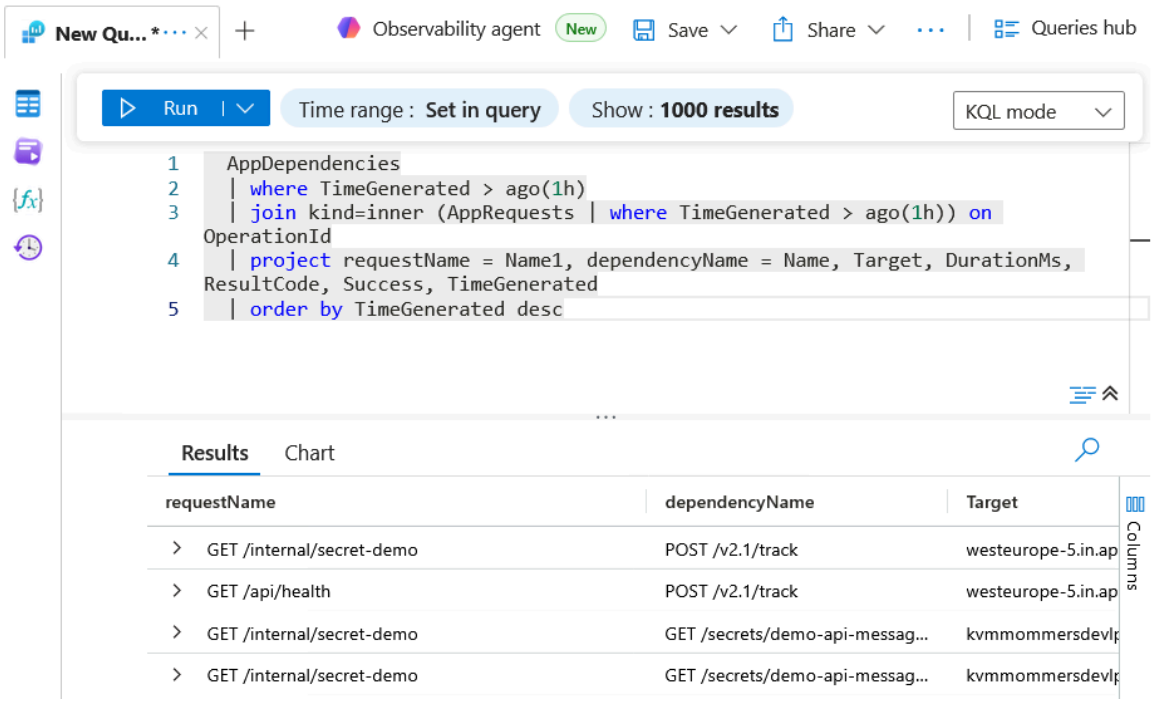
> **Note**
>
> If you don't see any Table Storage calls, check that the internal backend is properly instrumented and that you generated traffic that triggers Table Storage access (e.g., the `/api/data-demo` endpoint) and that you have the data in the account.

5. Trace a Request Through the Full Stack

5.1. Join Requests → Dependencies by OperationId

This is the most powerful query in the lab — it joins requests to their dependencies:

```
  AppDependencies
  | where TimeGenerated > ago(1h)
  | join kind=inner (AppRequests | where TimeGenerated > ago(1h)) on OperationId
  | project requestName = Name1, dependencyName = Name, Target, DurationMs,
ResultCode, Success, TimeGenerated
  | order by TimeGenerated desc
```



> **Question**
>
> Look at the joined results. For each request, what dependencies were triggered? Can you trace the path from the HTTP request through to Key Vault or Table Storage calls?

5.2. What If the Internal Backend Is Not Instrumented?

If the internal backend does not have the OpenTelemetry-based SDK installed, its calls to Key Vault and Table Storage will not appear in the `AppDependencies` table. The `OperationId` chain breaks at that point.

What would you see? The public backend's request would show dependencies for its own calls (e.g., to the internal backend), but you would not see the subsequent calls from the internal backend to Key Vault or Table Storage. This means:

- You could trace a request from the user to the internal backend, but not through it
- You would miss the full attack path if an attacker exploited the internal backend
- Your security investigations would have blind spots in the middle tier of your architecture

Question

The internal backend is already instrumented. If it were not, what specific calls would be missing from the `AppDependencies` table? Would this affect your ability to trace a request from the public backend through to Key Vault? Why or why not?

5.3. Trace the Secret Demo Request

The `/api/security/secret-demo` endpoint traces a request through the full stack: public backend → internal backend → Key Vault.

First, find a recent request to the endpoint:

```
AppTraces
| where TimeGenerated > ago(1h)
| where Message has "public-api-internal-backend-call"
| extend correlationId = tostring(Properties["correlationId"])
| where correlationId contains "01111111-1111-4111-b111-1111111115"
| project TimeGenerated, Message, OperationId, correlationId, Properties
| order by TimeGenerated desc
```

Use the operationId to inspect the related requests, traces, and dependencies:

```
let operationId = "87074d781622a288e71d46ecd0d3f843";
union isfuzzy=true withsource=source AppRequests, AppTraces, AppDependencies
| where TimeGenerated > ago(1h)
| where OperationId == operationId
| project
    Source,
    TimeGenerated,
    Name,
    Message,
    Target,
    ResultCode,
    Success,
    DurationMs,
    Properties
| order by TimeGenerated asc
```

You should see telemetry from different parts of the request flow. Look for:

- A request entry for the public backend endpoint: /api/security/secret-demo
- Trace entries from the application code
- Request or trace entries from the internal backend, depending on how it logs
- Dependency entries for outbound calls such as Key Vault or Table Storage

Warning

You should watchout with using telemetry auto detection for services like Key-Vault as Azure itself states that this is still best effort and can be quite troublesome at times. It would be better to have clear telementry in your own code.

6. Reflection Questions

Your answer:

1. A user sends a request without an `x-correlation-id` header. The public backend generates a UUID and uses it. Later, the user sends a second request with `x-correlation-id: user-request-abc`. Can you correlate these two requests? Why or why not?
2. An attacker sends 100 requests in 10 seconds, each with a different correlation ID. The application logs each request with its correlation ID. Can you still identify this as a single attack? What KQL query would you use against the `AppTraces` table?
3. Why is structured JSON logging more useful for security investigations than free-form text logging? Give a specific example using the data from this lab.

7. Summary

In this lab, you verified that the OpenTelemetry-based SDK (`@azure/monitor-opentelemetry`) is properly instrumented in both the public and internal backends. You generated traffic with custom correlation IDs (the `x-correlation-id` header) and traced those requests through the full application stack using Log Analytics. You queried the `AppRequests`, `AppTraces`, and `AppDependencies` tables to follow a single request from the public backend to the internal backend to Key Vault. You understand how correlation IDs let you trace a specific client request across services, and how `operation_Id` links all telemetry records (requests, traces, dependencies) together for end-to-end tracing.