

OpenTelemetry quickstart: your first GAS trace

Genero Application Server 6.01.00 and later · October 2026

This guide takes you from a standard GAS installation to a first trace on screen in about ten minutes. It covers the common case, and ends with the list of spans you can see in a trace.

What you get

With tracing on, every HTTP request handled by GAS becomes one distributed trace that follows the request through each process:

```
Browser (GBC) or web service client
├── GAS dispatcher      service gas-httpdispatch
│   ├── uaproxy / gwsproxy service gas-uaproxy / gas-gwsproxy
│   │   └── DVM (fglrun)  service fgl: 4GL function calls and SQL statements
```

Each process sends its own spans to an OpenTelemetry collector, which puts them together in one trace. You then see how long each step took: the dispatcher, the proxy, the 4GL functions and the SQL statements.

Before you start, note:

- GAS exports with **OTLP over HTTP, JSON encoding** (the `/v1/traces` endpoint, port `4318` by convention). OTLP over gRPC is not supported.
- Only traces are exported: no metrics and no logs.

Prerequisites

- Genero Application Server (GAS) **6.01.00 or later**.
- To get the spans of your 4GL programs: Genero BDL runtime (fglrun) **6.01.00 or later**. With an older runtime you still get the GAS spans, but no function or SQL spans.
- An OpenTelemetry collector or a tracing back end that accepts OTLP/HTTP. This guide uses [Jaeger](#), which you can run with Docker on your workstation. Any OTLP/HTTP back end works the same way (Grafana Tempo, an OpenTelemetry Collector, a vendor's endpoint, ...).

Step 1: start a collector

Start Jaeger:

```
docker run --rm --name jaeger -p 16686:16686 -p 4318:4318 jaegertracing/jaeger:2.20.0
```

- `4318` receives the traces (OTLP over HTTP).
- `16686` serves the Jaeger UI: open `http://localhost:16686`.

Jaeger keeps the traces in memory: they are lost when the container stops. This is fine for a first test; use a persistent back end for anything else.

If the collector runs on another machine, note its address: you need it in the next step, as `http://<collector-host>:4318`.

Step 2: turn tracing on

Tracing is **off by default**. Choose one of the options below.

Option A: everything, for a quick test

Start the dispatcher with tracing on:

```
httpdispatch -E res.otel.enabled=TRUE
```

When the collector is not on the GAS machine, give its address too:

```
httpdispatch -E res.otel.enabled=TRUE -E res.otel.endpoint=http://collector-host:4318
```

Give the **base** URL of the collector: GAS adds `/v1/traces` itself.

Option B: everything, permanently

Edit the resources in `$FGLASDIR/etc/as.xcf` (you may prefer to keep a copy of the original):

```
<RESOURCE Id="res.otel.enabled" Source="INTERNAL">TRUE</RESOURCE>
<RESOURCE Id="res.otel.endpoint" Source="INTERNAL">http://collector-host:4318</RESOURCE>
```

Then restart the dispatcher.

Keep the `<OPENTELEMETRY>` element under `<INTERFACE_TO_CONNECTOR>` in `as.xcf`. If you remove it, tracing is switched off completely, and none of the options below can turn it back on.

Option C: one application or service only (recommended)

Keep `res.otel.enabled` set to `FALSE` and, if needed, set `res.otel.endpoint` as in option B. Then turn tracing on just where you need it, in one of these ways:

- **For one session:** add `?otel=1` to the start URL. No configuration change is needed:

```
http://gas-host:6394/ua/r/myapp?otel=1
```

- **For every session of an application, from now on:** add an `<OPENTELEMETRY>` element before `<EXECUTION>` in its `.xcf` file:

```
<APPLICATION Parent="defaultgwc"
  xsi:noNamespaceSchemaLocation="http://www.4js.com/ns/gas/6.00/cfextwa.xsd">
  <OPENTELEMETRY Enabled="TRUE"/>
  <EXECUTION>
    <PATH>$(user.application.path)/myapp</PATH>
    <MODULE>myapp</MODULE>
  </EXECUTION>
</APPLICATION>
```

The same element works in a web service `.xcf` file.

- **For an application or service, on a running dispatcher, without editing any file:**

```
gasadmin application enable-otel myapp
gasadmin service enable-otel myservice
```

This applies to the sessions started after the command and is forgotten when the dispatcher restarts. Use `disable-otel` to turn it off again.

When several of these settings apply, the first one found in this list wins:

1. `?otel=1` or `?otel=0` in the start URL
2. `gasadmin ... enable-otel` or `disable-otel`
3. `<OPENTELEMETRY Enabled="...">` in the application or service `.xcf` file
4. `res.otel.enabled` in `as.xcf`

So `Enabled="FALSE"` in an application `.xcf` file also excludes that application when tracing is on for everything else.

Step 3: send a request and see the trace

Run your application from a browser (for example `http://gas-host:6394/ua/r/myapp?otel=1`), or call one of your web services:

```
curl http://gas-host:6394/ws/r/myservice/...
```

Then open the Jaeger UI at `http://localhost:16686`, choose the `gas-httpdispatch` service and click **Find Traces**. A web service call looks like this:

POST /ws/r/demo/Calculator	gas-httpdispatch
└─ gas.request.forward	gas-httpdispatch
└─ gwsproxy.request	gas-gwsproxy
└─ gwsproxy.dvm.acquire	gas-gwsproxy
└─ gwsproxy.request.receive	gas-gwsproxy
└─ dvm.request	fgl
└─ Calculator.Add	fgl
└─ gwsproxy.app.wait	gas-gwsproxy
└─ gwsproxy.response.send	gas-gwsproxy
└─ gas.app.wait	gas-httpdispatch
└─ gas.response.forward	gas-httpdispatch

and a user interface request like this:

POST /ua/sua/...	gas-httpdispatch
└─ gas.request.forward	gas-httpdispatch
└─ uaproxy.request	gas-uaproxy
└─ uaproxy.request.receive	gas-uaproxy
└─ dvm.request	fgl
└─ myapp.main	fgl
└─ myapp.load_customers	fgl
└─ OPEN c1	fgl db.query.text="SELECT ... WHERE id = ?"
└─ uaproxy.app.wait	gas-uaproxy
└─ uaproxy.response.send	gas-uaproxy
└─ gas.app.wait	gas-httpdispatch
└─ gas.response.forward	gas-httpdispatch

The very first request of an application (`GET /ua/r/myapp`) has no `fgl` spans: the program has not been started yet at that point. The program spans appear from the next requests on.

The `fastcgidispach` and `isapidispach` dispatchers report the same `gas-httpdispatch` service name.

If your web service client already sends a W3C `traceparent` header, GAS continues the client's trace: the GAS and program spans appear inside it.

Optional: write the spans to a file

Without a collector, you can check that spans are produced by writing them to a file, one JSON document per line:

```
GAS_OTEL_ENDPOINT=file:///tmp/traces.jsonl httpdispatch -E res.otel.enabled=TRUE
```

The `GAS_OTEL_ENDPOINT` environment variable takes precedence over `res.otel.endpoint`, which makes it handy for a quick test. Remember to unset it afterwards. A file endpoint is meant for tests only: it is never cleaned up.

Optional: send to an HTTPS collector

Give an `https://` endpoint. The collector certificate is checked against the system's certificate authorities (the Windows certificate store on Windows).

- If your company's own certificate authority signed the collector certificate, give its certificate (PEM file) in `res.otel.ca_file`:

```
httpdispatch -E res.otel.enabled=TRUE -E res.otel.endpoint=https://collector-host:4318 \
-E res.otel.ca_file=/etc/pki/my-company-ca.pem
```

If that file cannot be read, the dispatcher does not start.

- For a self-signed certificate, during a test only, set `res.otel.insecure` to `TRUE`: the certificate and the host name are then not checked at all. A warning is written to the dispatcher log.

What is recorded, and how to change it

These settings exist in `as.xcf` (as `res.otel.*` resources) and, as attributes of the `<OPENTELEMETRY>` element, in each application or service `.xcf` file:

Setting	Default	Effect
<code>Sanitize</code>	<code>TRUE</code>	Replaces the values written in SQL statements with <code>?</code> , so no data reaches the traces.
<code>CaptureValues</code>	<code>FALSE</code>	<code>TRUE</code> records the SQL parameter values and the function arguments and results. Only works with <code>Sanitize="FALSE"</code> .
<code>ContentMaxBytes</code>	<code>128</code>	How many bytes of each request and response the proxies record. <code>0</code> records none.
<code>ExcludeFunctions</code>	<code><builtin>.</code>	A regular expression: matching 4GL functions (<code>module.function</code>) are not recorded. <code>""</code> records every function. In XML, write <code>&lt;</code> and <code>&gt;</code> for <code><</code> and <code>></code> .

For example, to see the SQL values of one application during an investigation:

```
<OPENTELEMETRY Enabled="TRUE" Sanitize="FALSE" CaptureValues="TRUE"/>
```

Troubleshooting

Symptom	What to check
No trace at all	Tracing is on (step 2); the collector is reachable from the GAS machine on its port; <code>GAS_OTEL_ENDPOINT</code> is not set to another target; the endpoint is the base URL, without <code>/v1/traces</code> .
Only <code>gas-httpdispatch</code> spans	Tracing is not on for that application: check the order of the settings in step 2.
GAS spans but no <code>fgl</code> spans	The runtime is older than 6.01.00; or this is the first request of the application (see step 3).
Nothing exported over HTTPS	The collector certificate is not trusted: set <code>res.otel.ca_file</code> (see above).
Requests are slow when the collector has a problem	The spans are sent while the request is handled. A collector that accepts connections but does not answer delays each request by up to 2 seconds per process. Fix or stop the collector, or turn tracing off.
Non-ASCII text in SQL statements shows as <code>????</code>	Known limit: text that is not UTF-8 cannot be recorded yet, and is replaced.

In production

- Turn tracing on only where you need it (option C), rather than for everything.
- Keep `Sanitize="TRUE"` unless you must see the values, and turn it back on afterwards.
- Do not enable the `DEBUG` log category while tracing is on: it writes the full content of every trace to the log, which is large and may contain request data.
- Make sure the collector is available and answers quickly: see Troubleshooting.

Spans reference

These are the spans you can see in a trace, by the service that sends them.

Dispatcher (`gas-httpdispatch`)

Span	What it measures
<code><METHOD> <uri></code> (e.g. <code>GET /ua/r/myapp</code>)	The whole request, from its arrival in GAS to the end of the response. The root of the trace.
<code>gas.request.forward</code>	Forwarding the request to the proxy.
<code>gas.app.wait</code>	The wait for the first byte of the answer from the proxy: the time spent in the application.
<code>gas.response.forward</code>	Sending the response back to the client.

The root span records the HTTP method, path and status code, the client address, the request and response sizes, and the GAS session and application identifiers. Its status is `ERROR` when the response status is 500 or more.

User interface proxy (`gas-uaproxy`)

Span	What it measures
<code>uaproxy.request</code>	The request as handled by the proxy.
<code>uaproxy.request.receive</code>	Reading the request, up to handing it to the program.
<code>uaproxy.app.wait</code>	The wait for the program's answer.
<code>uaproxy.response.send</code>	Sending the answer back to the dispatcher.

Web service proxy (`gas-gwsproxy`)

Span	What it measures
<code>gwsproxy.request</code>	The service call as handled by the proxy.
<code>gwsproxy.dvm.acquire</code>	The wait for a program to take the call: a free program in the pool, or a new one being started. Its status is <code>ERROR</code> if the call is dropped meanwhile.
<code>gwsproxy.request.receive</code>	Reading the call and passing it to the program.
<code>gwsproxy.app.wait</code>	The wait for the service response.
<code>gwsproxy.response.send</code>	Sending the response back to the dispatcher.

The proxy spans record up to `ContentMaxBytes` bytes of the request and response, as the `content` attribute.

Program (fgl)

Span	What it measures
dvm.request	One user interface order, or one web service call, in the program.
dvm.startup	The program start, before its first order.
<module>.<function> (e.g. myapp.load_customers)	One 4GL function call, nested as the calls are. Functions that match ExcludeFunctions are left out.
<VERB> <target> (e.g. OPEN c1), <VERB> or sql	One SQL operation.

- Function spans record where the function is called from (module, function and line). With CaptureValues="TRUE" , they also record the arguments and the return values.
- SQL spans record the statement (db.query.text , with its values replaced by ? while Sanitize="TRUE"), the number of rows returned or changed, the SQL status code and the 4GL line. Their status is ERROR when the SQL status code is negative. With CaptureValues="TRUE" and Sanitize="FALSE" , they also record the parameter and result values.
- User interface events (for example ON ACTION) are recorded as events on the dvm.request span, not as separate spans.