OpenTelemetry

Version JS SDK 0.300 / API 1.10 / Collector 0.164 · verified 2026-11-02

Signals, the API/SDK split, Node auto-instrumentation, manual spans, traceparent propagation, the Collector pipeline and sampling.

Contents

Mental model 2

Signals, API and SDK 2

Bootstrapping a service 3

Manual spans and attributes 4

Configuration and resources 5

Context propagation 6

The Collector pipeline 7

Sampling and cost control 9

Further reading 10

Mental model

A trace, its metrics and its logs are three views of the same request, joined by a trace ID and span ID that travel with the request in a `traceparent` header. Your code talks only to the vendor-neutral API; the SDK you install at startup decides what is recorded, sampled and exported. Without an SDK the API is a silent no-op, so “no data” almost always means the SDK started too late or not at all.

Signals, API and SDK

Piece	What it is
Traces	Spans: timed operations with parent links
Metrics	Counters, histograms, gauges
Logs	Records stamped with trace and span IDs
API	`@opentelemetry/api` — what code calls
SDK	Sampling, processing, export

Libraries depend only on the API, so they can be instrumented once and used anywhere. The application owner installs the SDK and picks exporters. Every span, metric and log carries a **resource** describing who emitted it — `service.name` above all.

```
npm install @opentelemetry/api \
  @opentelemetry/sdk-node \
  @opentelemetry/auto-instrumentations-node \
  @opentelemetry/exporter-trace-otlp-http
```

Bootstrapping a service

Stack	How
Node, in code	`NodeSDK` loaded with `node --import`
Node, zero-code	`--require …/register` via `NODE_OPTIONS`
ASP.NET Core	`AddOpenTelemetry().WithTracing(…)`
.NET, zero-code	OpenTelemetry .NET Automatic Instrumentation

Auto-instrumentation patches popular libraries (HTTP, Express, database drivers) so inbound and outbound calls become spans with no code changes. With an exporter and no endpoint, the OTLP HTTP exporter sends to `http://localhost:4318/v1/traces`.

```
// instrumentation.ts
import { NodeSDK } from "@opentelemetry/sdk-node";
import { getNodeAutoInstrumentations } from
  "@opentelemetry/auto-instrumentations-node";
import { OTLPTraceExporter } from
  "@opentelemetry/exporter-trace-otlp-http";

const sdk = new NodeSDK({
  serviceName: "checkout",
  traceExporter: new OTLPTraceExporter(),
  instrumentations: [getNodeAutoInstrumentations()],
});
```

```
sdk.start();
process.on("SIGTERM", () => sdk.shutdown());
```

> **Gotcha:** The SDK must load *before* your app imports anything it patches. Run `node --import ./instrumentation.mjs app.js`; importing it from inside `app.js` is too late, and the libraries stay uninstrumented with no error.

Manual spans and attributes

Call	Effect
`trace.getTracer(name)`	Tracer for your module
`startActiveSpan(name, fn)`	Child span, current in `fn`
`span.setAttribute(k, v)`	Searchable key/value
`span.recordException(e)`	Adds an exception event
`span.setStatus({ code })`	Marks `ERROR` or `OK`
`span.end()`	Records the span

Add spans only for work auto-instrumentation cannot see, such as business steps. Name attributes after the semantic conventions where one exists (`http.request.method`, `http.route`, `http.response.status_code`) so backends recognise them; prefix your own with an app namespace.

```
import * as api from "@opentelemetry/api";
const tracer = api.trace.getTracer("checkout");
```

```
const { ERROR } = api.SpanStatusCode;

function charge(id: string, pay: () => void) {
  tracer.startActiveSpan("charge", (span) => {
    span.setAttribute("app.order.id", id);
    try { pay(); } catch (err) {
      span.recordException(err as Error);
      span.setStatus({ code: ERROR });
      throw err;
    } finally { span.end(); }
  });
}
```

> **Gotcha:** A span that is never ended is never exported, and `recordException` alone leaves the status unset. End spans in `finally`, and set `ERROR` explicitly.

Configuration and resources

Variable	Purpose
`OTEL_SERVICE_NAME`	Sets `service.name`
`OTEL_RESOURCE_ATTRIBUTES`	Extra `k=v,k=v` resource attributes
`OTEL_EXPORTER_OTLP_ENDPOINT`	Base URL for all signals
`OTEL_EXPORTER_OTLP_PROTOCOL`	`http/protobuf` or `grpc`
`OTEL_TRACES_SAMPLER`	Default `parentbased_always_on`
`OTEL_PROPAGATORS`	Default `tracecontext,baggage`

Every SDK reads the same environment variables, so you can configure Node and .NET services identically from a deployment manifest. `OTEL_SERVICE_NAME` wins over a `service.name` in `OTEL_RESOURCE_ATTRIBUTES`.

```
export OTEL_SERVICE_NAME=checkout
export OTEL_RESOURCE_ATTRIBUTES=\
service.version=1.4.2,\
deployment.environment.name=prod
export OTEL_EXPORTER_OTLP_ENDPOINT=\
http://collector:4318
# traces go to http://collector:4318/v1/traces
```

> **Gotcha:** Only the generic `OTEL_EXPORTER_OTLP_ENDPOINT` gets `/v1/traces` appended. A per-signal variable such as `OTEL_EXPORTER_OTLP_TRACES_ENDPOINT` is used verbatim, so it must include the path.

Context propagation

Field of `traceparent`	Size
version	2 hex digits (`00`)
trace-id	32 hex digits
parent-id (span ID)	16 hex digits
trace-flags	2 hex digits; `01` = sampled

The W3C Trace Context `traceparent` header carries the trace across process boundaries; the default propagators also carry `tracestate` and `baggage`. Instrumented HTTP clients and servers inject and extract it for you. For transports that are not instrumented — a queue message, a custom RPC — inject into a carrier object yourself and extract on the other side.

```
import { context, propagation } from
  "@opentelemetry/api";

const headers: Record<string, string> = {};
propagation.inject(context.active(), headers);
// headers.traceparent is now
//   "00-<trace-id>-<span-id>-01"

const ctx = propagation.extract(
  context.active(),
  headers,
);
```

The Collector pipeline

Component	Job
Receivers	Accept data (`otlp` on 4317 gRPC, 4318 HTTP)
Processors	Limit, filter, enrich, sample
Exporters	Send on (`otlp_grpc`, `otlp_http`, ...)

Component	Job
`service.pipelines`	Wires them per signal

Services send OTLP to a Collector, and the Collector holds credentials, retries, batching and routing — so a backend change is a config change, not a redeploy. A component does nothing until a pipeline lists it. Batching now lives on the exporter's `sending_queue`.

```
receivers:
  otlp:
    protocols:
      http: { endpoint: 0.0.0.0:4318 }
exporters:
  otlp_grpc:
    endpoint: backend.example:4317
    sending_queue:
      batch:
service:
  pipelines:
    traces:
      receivers: [otlp]
      exporters: [otlp_grpc]
```

> **Gotcha:** The `otlp_grpc` exporter requires TLS by default. Sending to a plaintext in-cluster endpoint fails until you set `tls: { insecure: true }`. The old names `otlp` and `otlphttp` are deprecated aliases.

Sampling and cost control

Technique	Where	Decides on
`traceidratio`	SDK (head)	Trace ID only
`parentbased_*`	SDK (head)	Follows the caller's flag
`tail_sampling`	Collector	The whole trace
`filter`	Collector	Drop by OTTL condition

Head sampling is cheap but blind: it cannot keep "all traces with an error". Tail sampling can, at the cost of buffering whole traces in memory. Put `memory_limiter` first in every pipeline, then `tail_sampling`; a trace is kept if any policy says sample. On metrics, cost is cardinality — never put user IDs or raw URLs in attributes.

```
processors:
  memory_limiter:
    check_interval: 1s
    limit_percentage: 80
  tail_sampling:
    decision_wait: 10s
    policies:
```

```
    - name: keep-errors
      type: status_code
      status_code: { status_codes: [ERROR] }
    - name: ten-percent
      type: probabilistic
      probabilistic:
        sampling_percentage: 10
```

> **Warning:** Tail sampling needs every span of a trace on the same Collector instance. Behind a plain load balancer it silently makes decisions on partial traces; route by trace ID with the `loadbalancing` exporter in a first Collector tier.

Further reading

- OpenTelemetry documentation
- Node.js getting started
- .NET getting started
- Collector configuration
- Sampling concepts
- Semantic conventions
- W3C Trace Context