Monitoring a PHP application with OpenTelemetry

Enrico Zimuel, Principal Software Engineer

May 19, 2023 - phpDay Verona (Italy)

Summary

- Observability
- OpenTelemetry
- Signals: traces, metrics, log, baggage
- Collector
- Context propagation
- How to instrument a PHP application
- Manual instrumentation
- Automatic instrumentation

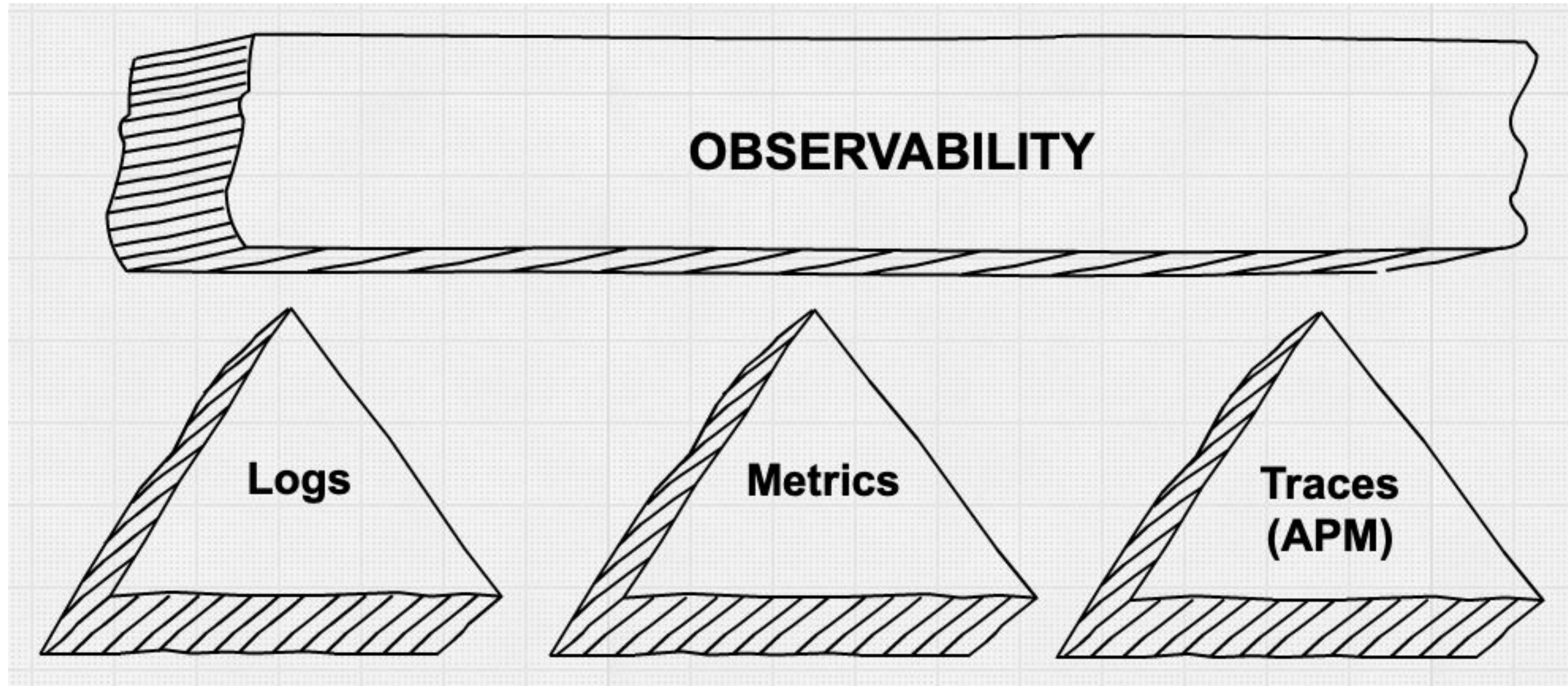
elastic

Observability

- **Observability** lets us understand a system from the outside
- We can observe the output of a system but this is not enough for understanding what is going on inside
- We need to **instrument** our application to emit **signals**
- Monitoring is not observability:
  - “Monitoring tells you whether a system is working, observability lets you ask why it isn't working” Baron Schwartz

elastic

Three pillars of observability



Source: Observability with the Elastic Stack

elastic

OpenTelemetry

- OpenTelemetry also known as **OTel** for short, is a vendor-neutral open-source Observability framework for instrumenting generating, collecting, and exporting telemetry data such as **traces**, **metrics**, **logs**
- Cloud Native Computing Foundation (CNCF) incubating project
- Natively supported by multiple vendors (including Elastic)

Signals

- **Signals** are the different types of data sent by an application to inform about the execution
- An application can emit the following signals:
  - **Traces**
  - **Metrics**
  - **Logs**

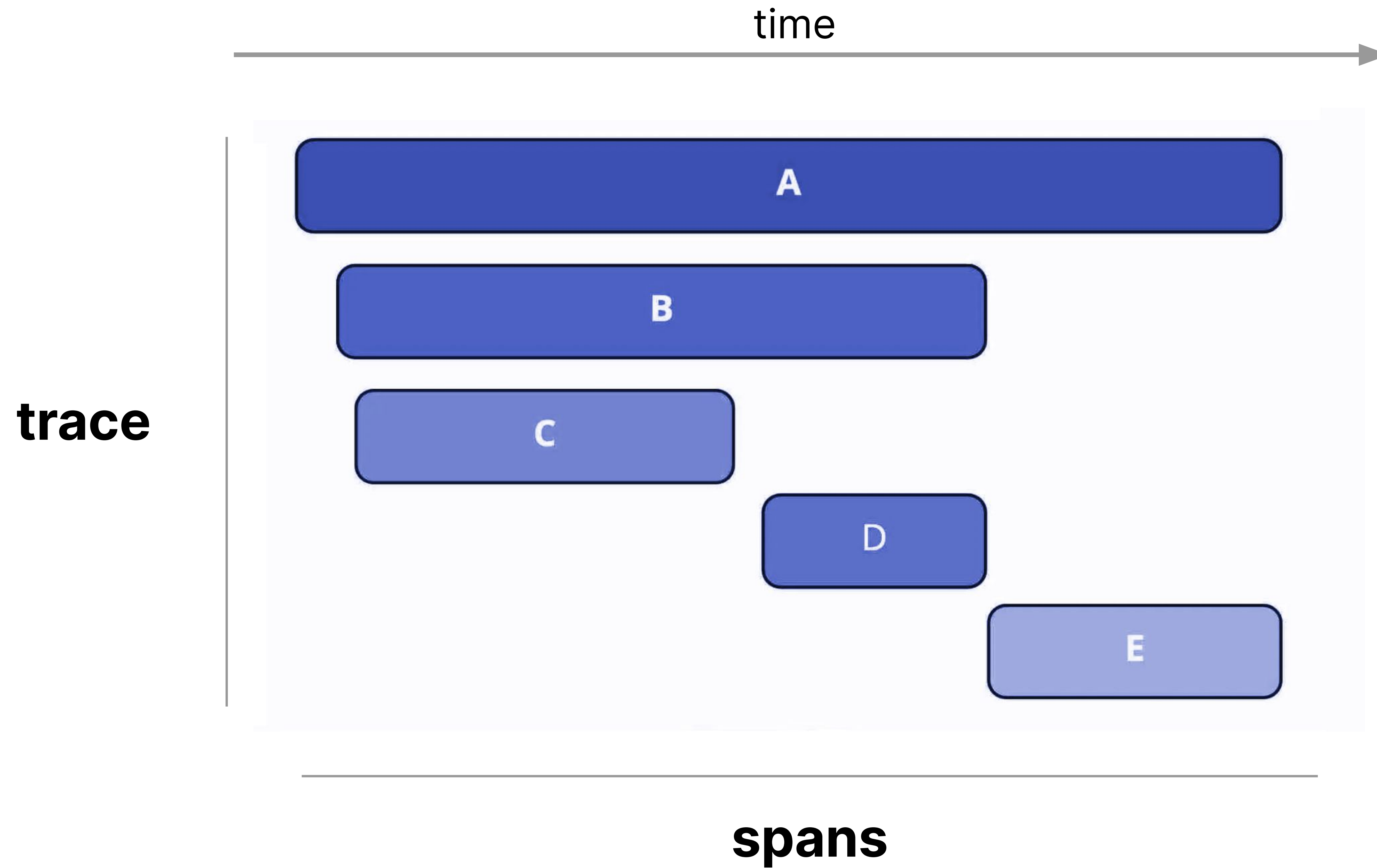
elastic

Traces

- A trace is a collection of information within a time frame
- A trace contains one or more **span**
- A span represents a unit of work or operation
- Spans are the building blocks of traces
- A span contains the following information:
  - Name, span ID, trace ID
  - Parent span ID (empty for root spans)
  - Start and End Timestamps
  - Span Context
  - Attributes
  - Events, Links, Status

elastic

Trace and spans



elastic

Metrics

- A **metric** is a measurement about a service, captured at runtime
- Application and request metrics are important indicators of availability and performance
- Custom metrics can provide insights into how availability indicators impact user experience or the business
- Collected data can be used to alert of an outage or trigger scheduling decisions to scale up a deployment automatically upon high demand.

elastic

Different metrics

- **Counter**: A value that accumulates over time
- **Asynchronous Counter**: same as the Counter, but is collected once for each export.
- **UpDownCounter**: A value that accumulates over time, but can also go down again.
- **Asynchronous UpDownCounter**: Same as the UpDownCounter, but is collected once for each export.
- **Gauge**: Measures a current value at the time it is read (asynchronous)
- **Histogram**: A histogram is a client-side aggregation of values, e.g., request latencies (e.g., How many requests take fewer than 1s?)

elastic

Logs

- A **log** is a timestamped text record, structured or unstructured, with metadata
- While logs are an independent data source, they may also be attached to spans
- In OpenTelemetry, any data that is not part of a distributed trace or a metric is a log

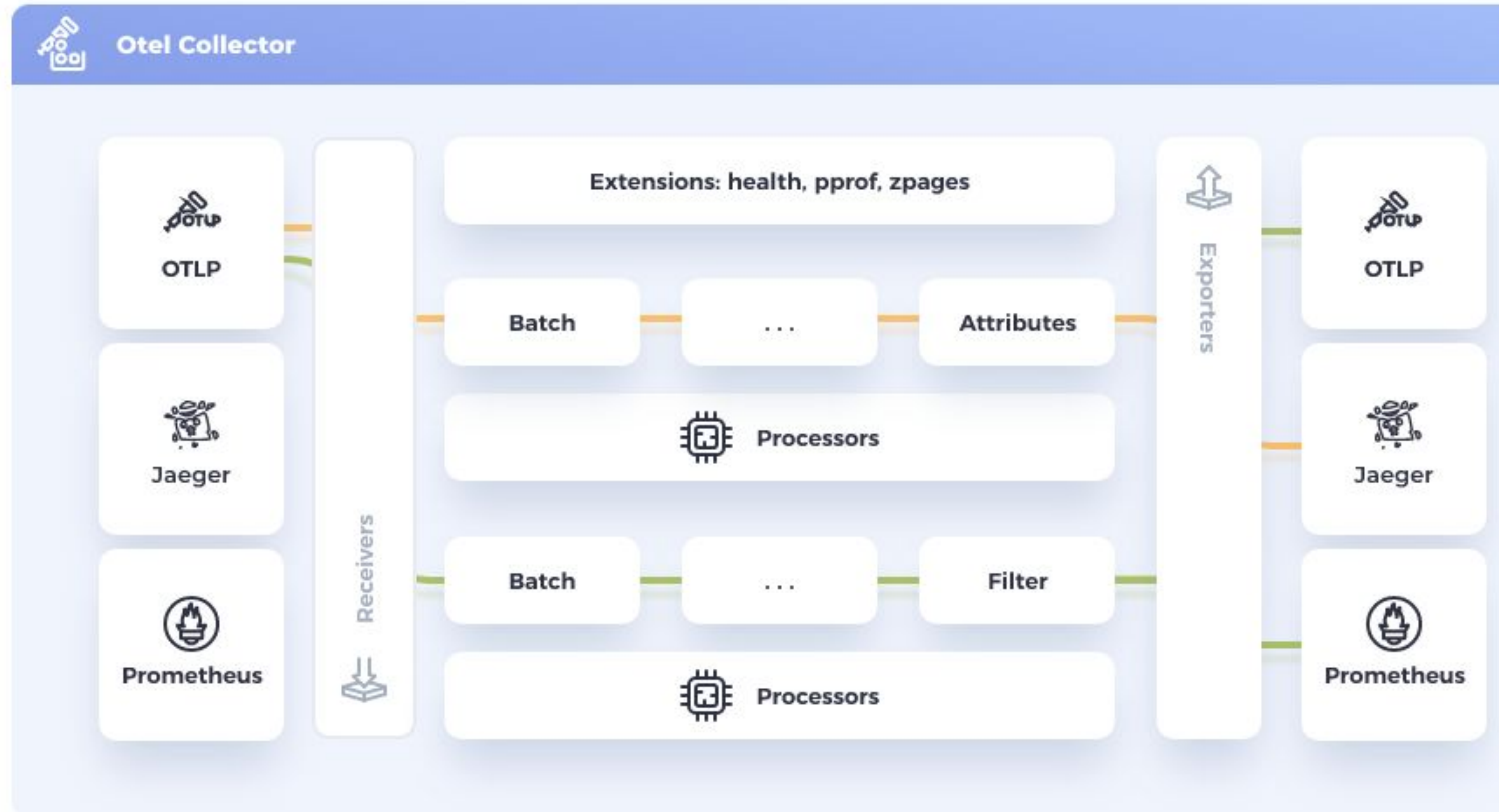
elastic

Collector

- **OTel collector** is a vendor-agnostic implementation of how to receive, process and export telemetry data
- It removes the need to run, operate, and maintain multiple agents/collectors
- Designed to scale and supports open source observability data formats sending to one or more open source or commercial back-ends
- The local Collector endpoint (localhost:4317/8) is the default location to which Otel SDK libraries export their telemetry data
- open-telemetry/opentelemetry-collector written in Go

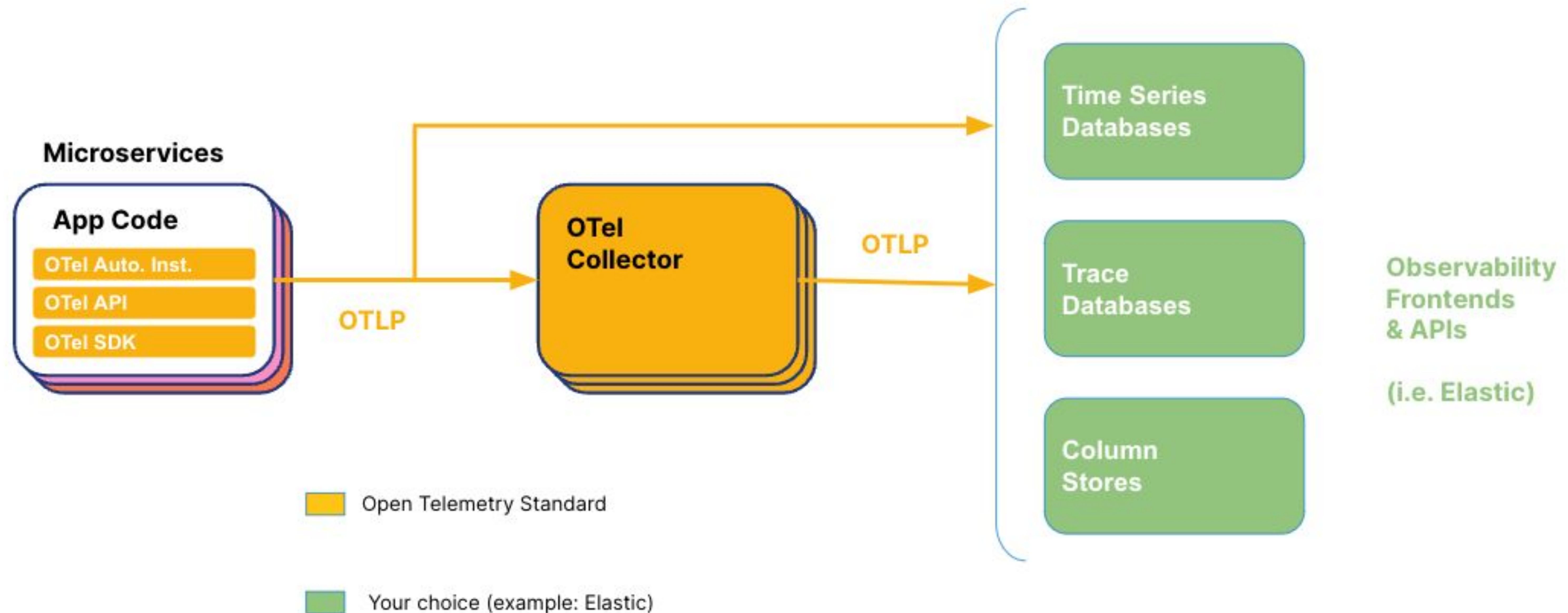
elastic

OTel collector diagram



elastic

OTel collector and back-end



elastic

Collector configuration

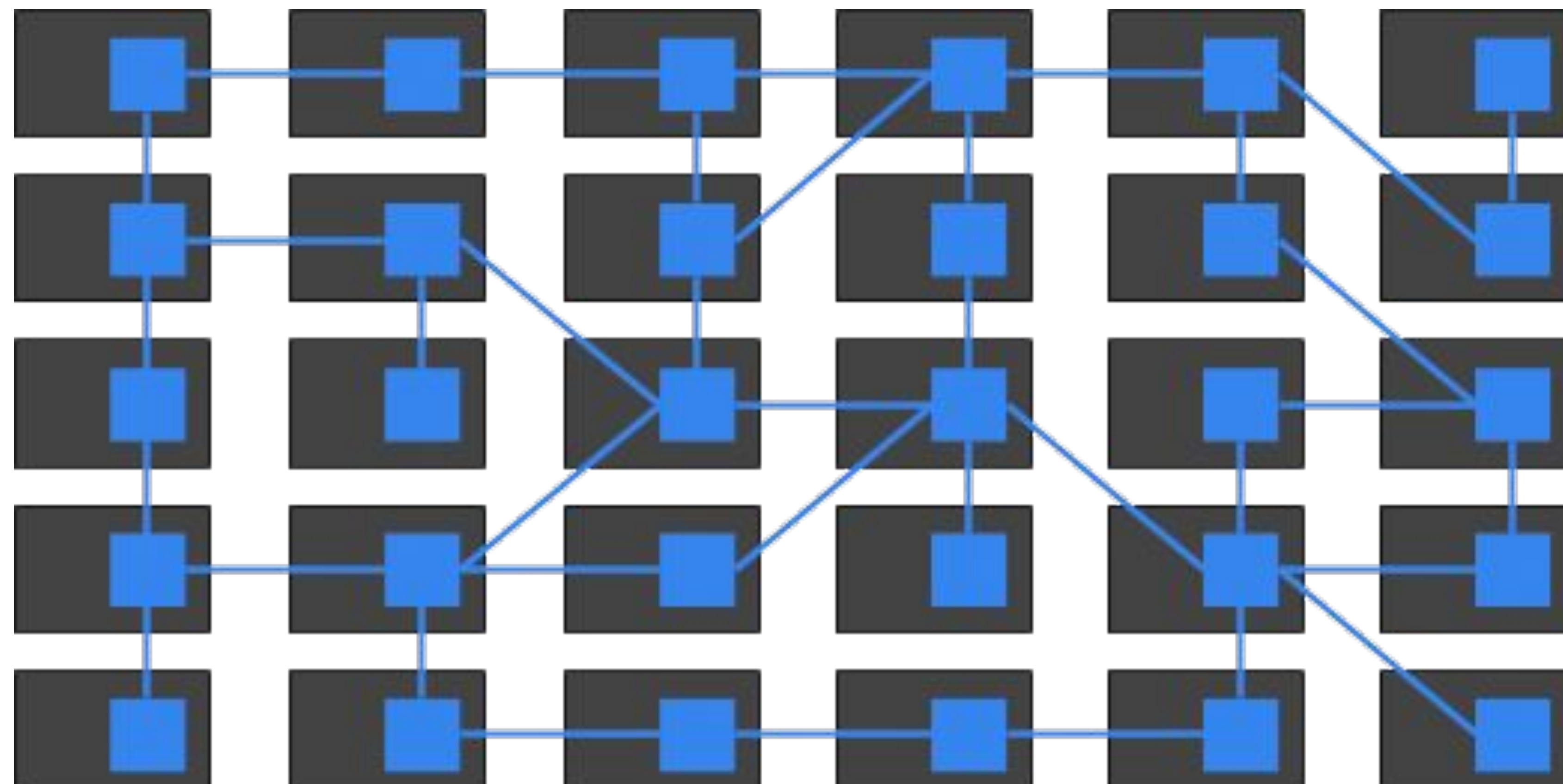
- OTel collector can be configure using a YAML file

```yaml
receivers:
 otlp:
   protocols:
     grpc:
     http:

processors:
 batch:

exporters:
 otlp:
   endpoint: otelcol:4317
 otlp/2:
   endpoint: otelcol2:4317
...
```

elastic

Microservices observability

- How to observe an application in a microservice/distributed architecture?



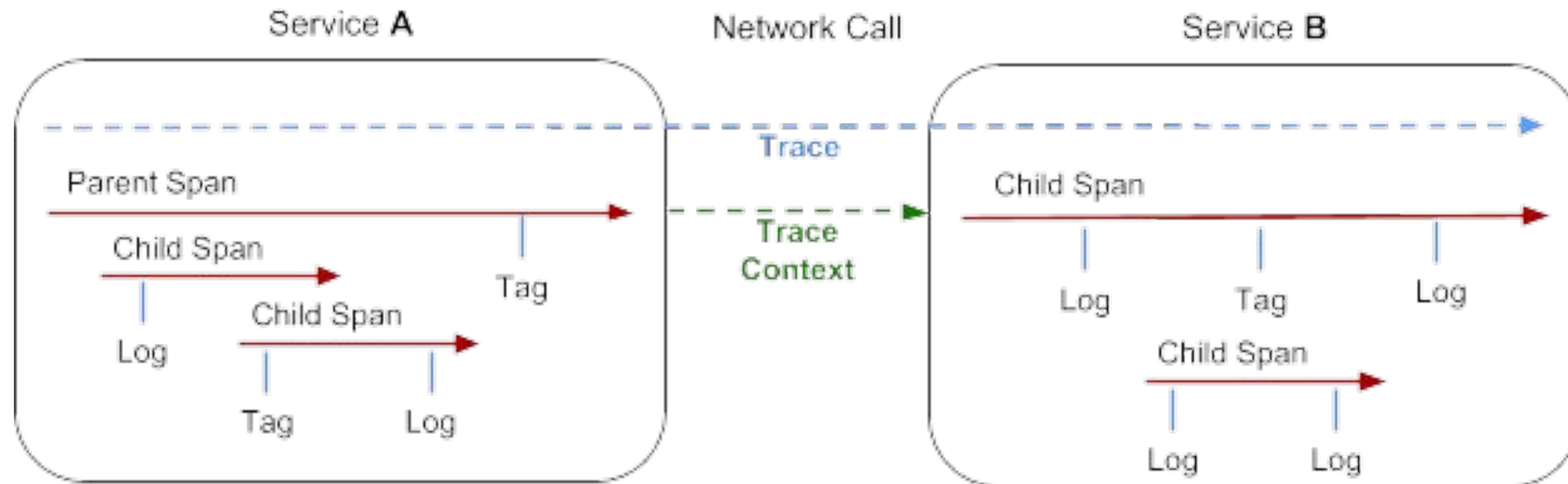
elastic

Context propagation

- Context propagation is the core concept that enables **Distributed Tracing**
- Spans can be correlated with each other and assembled into a trace
- Context Propagation is defined by two sub-concepts: **Context** and **Propagation**
  - **Context**: an object that contains the information for the sending and receiving service to correlate one span with another and associate it with the trace overall
  - **Propagation**: is the mechanism that moves Context between services and processes. It uses W3C TraceContext

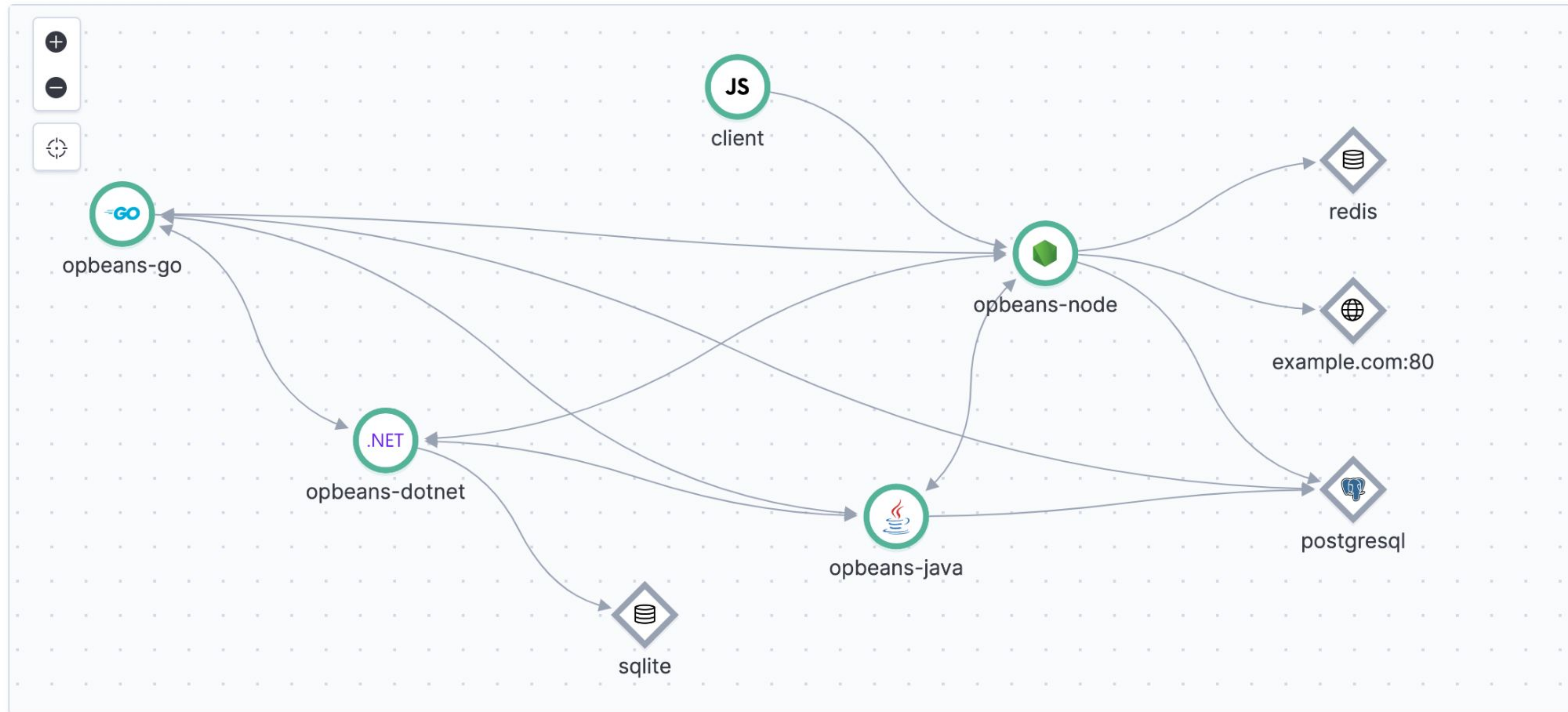
elastic

W3C TraceContext



elastic

Elastic Service Map



elastic

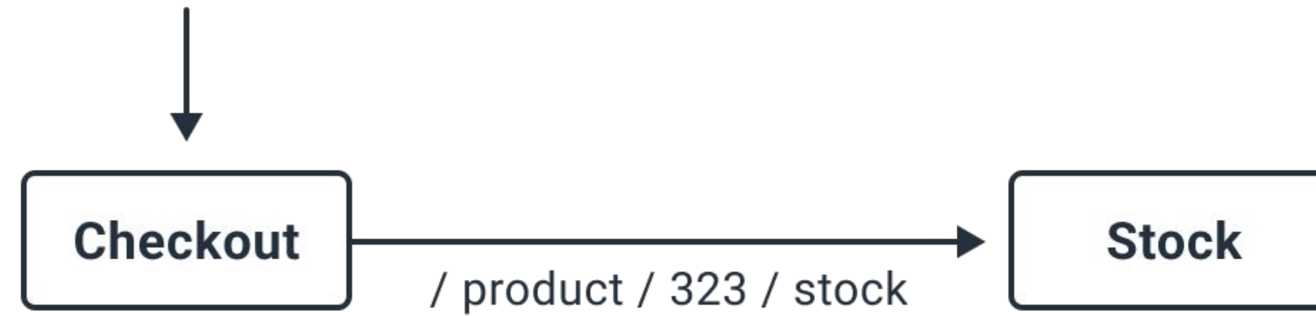
Baggage

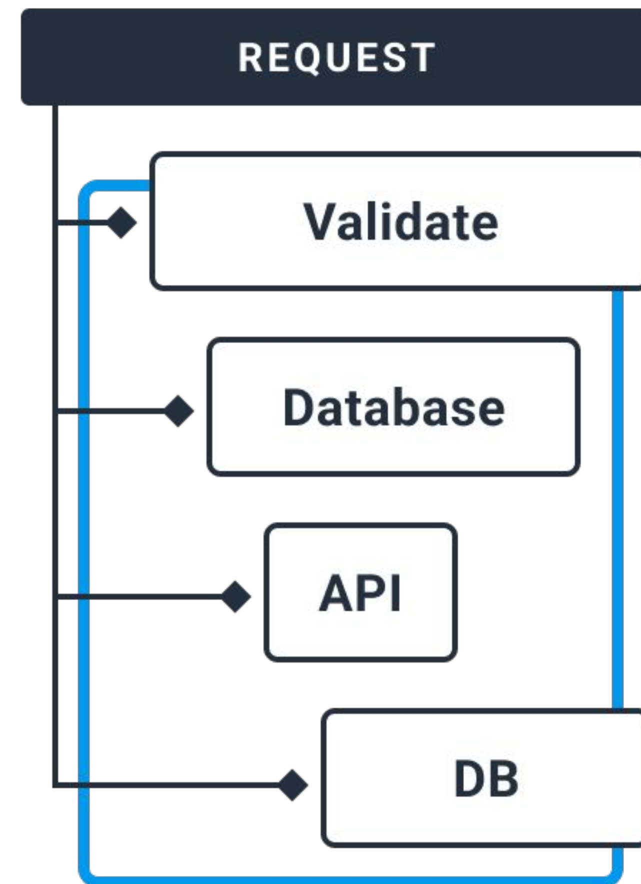
- **Baggage** is contextual information that’s passed between spans
- It’s a key-value store that resides alongside span context in a trace, making values available to any span created within that trace
- OpenTelemetry uses **Context Propagation** to pass Baggage around
- Baggage should be used for data that you’re okay with potentially exposing to anyone who inspects your network traffic

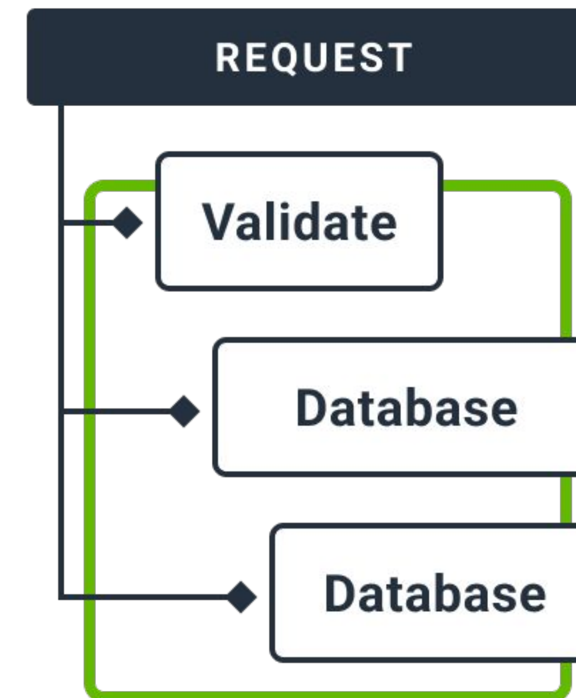
elastic

Baggage: example

/ account / 123 / order / 456







ACCOUNT ID = Path.Segment[1]

ACCOUNT ID = ??

OTLP

- **OpenTelemetry Protocol (OTLP)** describes the encoding, transport, and delivery mechanism of telemetry data between telemetry sources, intermediate nodes such as collectors, and telemetry backends
- It supports the following transports:
  - **OTLP/gRPC**, gRPC and HTTP/1.1 transports and specifies Protocol Buffers schema that is used for the payloads
  - **OTLP/HTTP**, use HTTP/2 or HTTP/1.1 and Protobuf payloads encoded either in binary format or in JSON format

elastic

OTel and PHP

elastic

OTel and PHP

- OTel provides a PHP SDK open-telemetry/opentelemetry-php
- Contains:
  - API interfaces for OTel implementation
  - Library for manual instrumentation (PHP 7.4+)
  - PHP extensions for auto-instrumentation (PHP 8.0+)
  - Exporters (for sending signals to different backends)
  - Auto-instrumentation modules (eg. WordPress, Laravel)
- Supports:
  - Traces (beta)
  - Metrics (beta)
  - Logs (alpha)

elastic

Requirements

- The OTel for PHP uses HTTP factories (PSR-17) and php-http/async-client
- We need to choose an async HTTP client, for instance:
  - composer require php-http/guzzle7-adapter
- PHP extensions:
  - ext-grpc, required for the OTLP exporter
  - ext-mbstring, better performance for byte string
  - ext-zlib, compress the exported data
  - ext-ffi, Fiber based context storage
  - ext-protobuf, significant performance improvement for OTLP

elastic

Manual instrumentation

- Install OTel SDK (enable "minimum-stability": "beta" in composer):
  - composer require open-telemetry/sdk
- Choose an **Exporter**
- Create a **TracerProvider**
- Create a **rootSpan**
- Create **spans**, **metrics** and **logs**

elastic

Example: TraceProvider with console exporter

```
use OpenTelemetry\SDK\Trace\SpanExporter\ConsoleSpanExporterFactory;
use OpenTelemetry\SDK\Trace\SpanProcessor\SimpleSpanProcessor;
use OpenTelemetry\SDK\Trace\TracerProvider;

$tracerProvider =  new TracerProvider(
   new SimpleSpanProcessor(
       (new ConsoleSpanExporterFactory())->create()
   )
);
$tracer = $tracerProvider->getTracer('io.opentelemetry.contrib.php');
$rootSpan = $tracer->spanBuilder('root')->startSpan();
$rootScope = $rootSpan->activate();

// create spans, metrics, logs

$rootSpan->end();
$rootScope->detach();
```

elastic

Example: TraceProvider with OTel exporter

```php
use OpenTelemetry\Contrib\Otlp\OtlpHttpTransportFactory;
use OpenTelemetry\Contrib\Otlp\SpanExporter;
use OpenTelemetry\SDK\Trace\SpanProcessor\SimpleSpanProcessor;
use OpenTelemetry\SDK\Trace\TracerProvider;

$transport = (new OtlpHttpTransportFactory())->create(
   'http://collector:4318/v1/traces',
   'application/x-protobuf'
);
$exporter = new SpanExporter($transport);
$tracerProvider =  new TracerProvider(
   new SimpleSpanProcessor(
       $exporter
   )
);
```

elastic

Example: Span

```
$span = $tracer->spanBuilder("my span")->startSpan();

// Make the span the current span
try {
    $scope = $span->activate();
    // In this scope, the span is the current/active span
} finally {
    $span->end();
    $scope->detach();
}
```

elastic

Example: Nested Span

```php
$parentSpan = $tracer->spanBuilder("parent")->startSpan();
$scope = $parentSpan->activate();
try {
   $child = $tracer->spanBuilder("child")->startSpan();
   //do stuff
   $child->end();
} finally {
   $parentSpan->end();
   $scope->detach();
}
```

elastic

Example: Metric

```
$reader = new ExportingReader((new ConsoleMetricExporterFactory())->create());
$meterProvider = MeterProvider::builder()
    ->addReader($reader)
    ->build();

$up_down = $meterProvider
    ->getMeter('my_up_down')
    ->createUpDownCounter('queued', 'jobs', 'The number of jobs enqueued');

//jobs come in
$up_down->add(2);
//job completed
$up_down->add(-1);
//more jobs come in
$up_down->add(2);

$meterProvider->forceFlush();
```

elastic

Example: Log

- OpenTelemetry can be configured to use a PSR-3 logger to log information about OpenTelemetry, including errors and warnings about misconfigurations or failures exporting data:

```
use OpenTelemetry\API\Common\Log\LoggerHolder;


$logger = new Psr3Logger(LogLevel::INFO);
LoggerHolder::set($logger);
```

elastic

Auto-instrumentation

- Install open-telemetry/opentelemetry-php-instrumentation ext:
  - **Pecl**:
    - pecl install opentelemetry-beta
  - **Pickle**:
    - php pickle.phar install --source https://github.com/open-telemetry/opentelemetry-php-instrumentation.git#1.0.0beta5
  - **Docker**:
    - install-php-extensions opentelemetry

elastic

Example

```
OpenTelemetry\Instrumentation\hook(
    'class': DemoClass::class,
    'function': 'run',
    'pre': static function () use ($tracer) {
        // pre code here
     },
    'post': static function () use ($tracer) {
        // post code here
    }
);
$demo = new DemoClass();
$demo->run();
```

elastic

Example: pre

```
'pre': static function (DemoClass $demo, array $params, string $class,
       string $function, ?string $filename, ?int $lineno) use ($tracer) {
       static $instrumentation;
       $instrumentation ??= new CachedInstrumentation('example');
       $span = $instrumentation->tracer()->spanBuilder($class)->startSpan();
       Context::storage()->attach($span->storeInContext(Context::getCurrent()));
}
```

elastic

Example: post

```
'post': static function (
    DemoClass $demo,
    array $params,
    $returnValue,
    ?Throwable $exception) use ($tracer) {
        $scope = Context::storage()->scope();
        $scope->detach();
        $span = Span::fromContext($scope->context());
        if ($exception) {
            $span->recordException($exception);
            $span->setStatus(StatusCode::STATUS_ERROR);
        }
        $span->end();
    }
```

elastic

References

- Bahubali Shetti, Independence with OpenTelemetry on Elastic
- Neha Duggal, Elastic introduces OpenTelemetry integration
- Elastic Observability and Security Teams, Elastic Common Schema and OpenTelemetry — A path to better observability and security with no vendor lock-in
- David Hope, Monitor OpenAI API and GPT models with OpenTelemetry and Elastic
- Ty Bekiares, Modern observability and security on Kubernetes with Elastic and OpenTelemetry
- Adam Quan, Distributed tracing, OpenTracing, and Elastic APM, webinar

elastic

Thanks!

More information about OpenTelemetry and the Elastic initiative about OTel

Contacts: enrico.zimuel@elastic.co

elastic