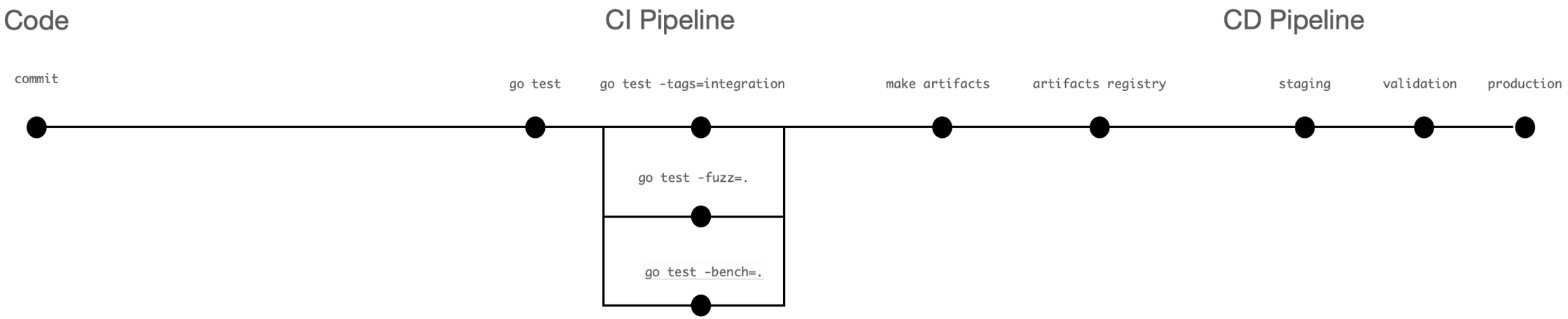
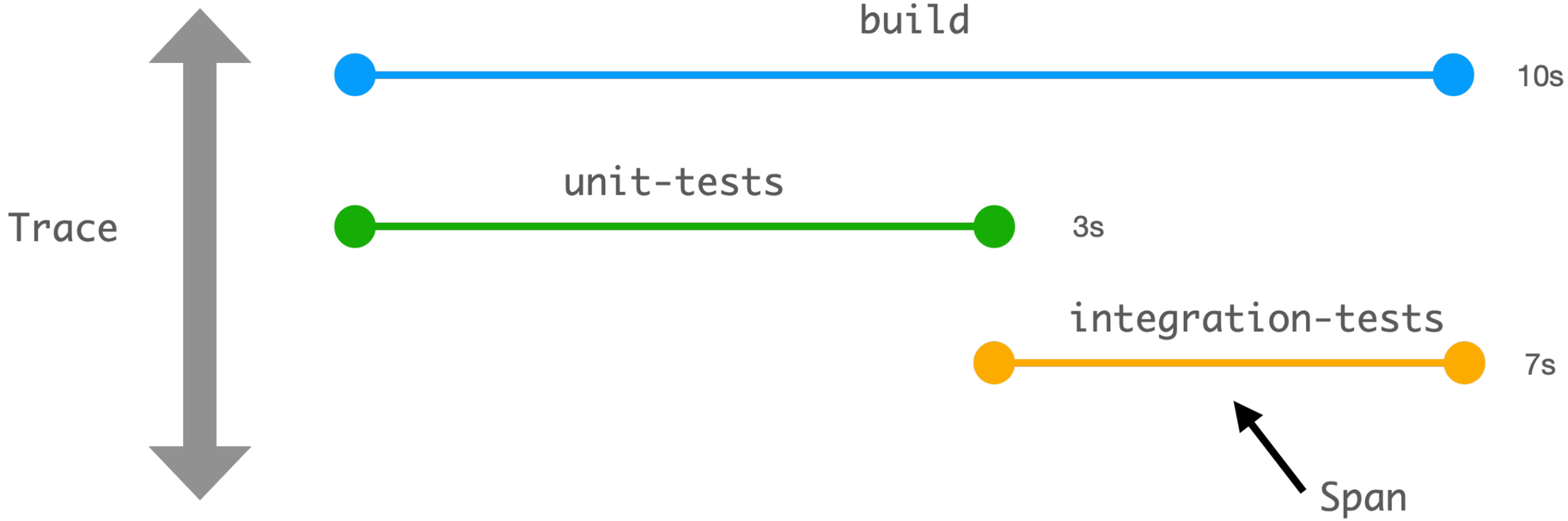
OpenTelemetry as best way how to instrument your CICD pipeline

Pipeline


Code
CI Pipeline
CD Pipeline
commit
go test
go test -tags=integration
make artifacts
artifacts registry
staging
validation
production
go test -fuzz=.
go test -bench=.

Distributed Tracing in 30 Seconds

- A span measures a unit of work in a service
- A trace is collection of spans



```
{
  "name": "build abtris/www.prskavec.net-Build_index-136-1",
  "context": {
    "trace_id": "abtris/www.prskavec.net-Build_index-136-1",
    "span_id": "abtris/www.prskavec.net-Build_index-136-1"
  },
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "ci.host": "hostname",
    "github.repository": "github.com/abtris/www.prskavec.net",
    "github.run_id": "6285578193"
  },
  "events": [
    {
      "name": "Hello Cloud Native!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```

Automatic vs Manual instrumentation

1. Automatic instrumentation where you don't have to make any changes in your application code to send telemetry data.
2. Manual instrumentation, where you create telemetry data by creating traces and metric events using the tracer and meter objects.

→ OpenTelemetry instumentation

→ Auto-instrumentation - supported by .Net, Java, JS, PHP, Python, Go in development

Tooling for instrumentation CI

- Trace
- Honeycomb buildevents
  - Create a GitHub Actions Workflow to Optimize Delivery Pipeline Performance With Honeycomb- great example how trace flaky tests via tracing
- Equinix-labs Otel CLI
- Jenkins OpenTelemetry plugin
- otel-desktop-viewer

Tooling for CD

- Keptn
- Making ANY K8s Deployment OBSERVABLE
- If you deploy with ArcoCD, Flux, GitLab, kubectl, etc. we provide you:
  - Automated App-Aware DORA metrics (OTel Metrics)
  - Troubleshoot failed deployments (OTel Traces)
  - Trace deployments from Git to cloud (traces across stages)
- Tracetest

Trace

Trace by Andy Davies

https://github.com/Pondidum/Trace/

- support for Grouping, Parallelism
- recommend read his post Tracing: structured logging, but better in every way

Usage

1. start a trace `export TRACEPARENT="$(trace start "some-name")`
2. start groups: `group=$(trace group start "some-name")`
3. run processes inside a group: `trace task "${group}" -- some command here`
4. finish the group `trace group finish "${group}"`
5. finish the trace `trace finish`

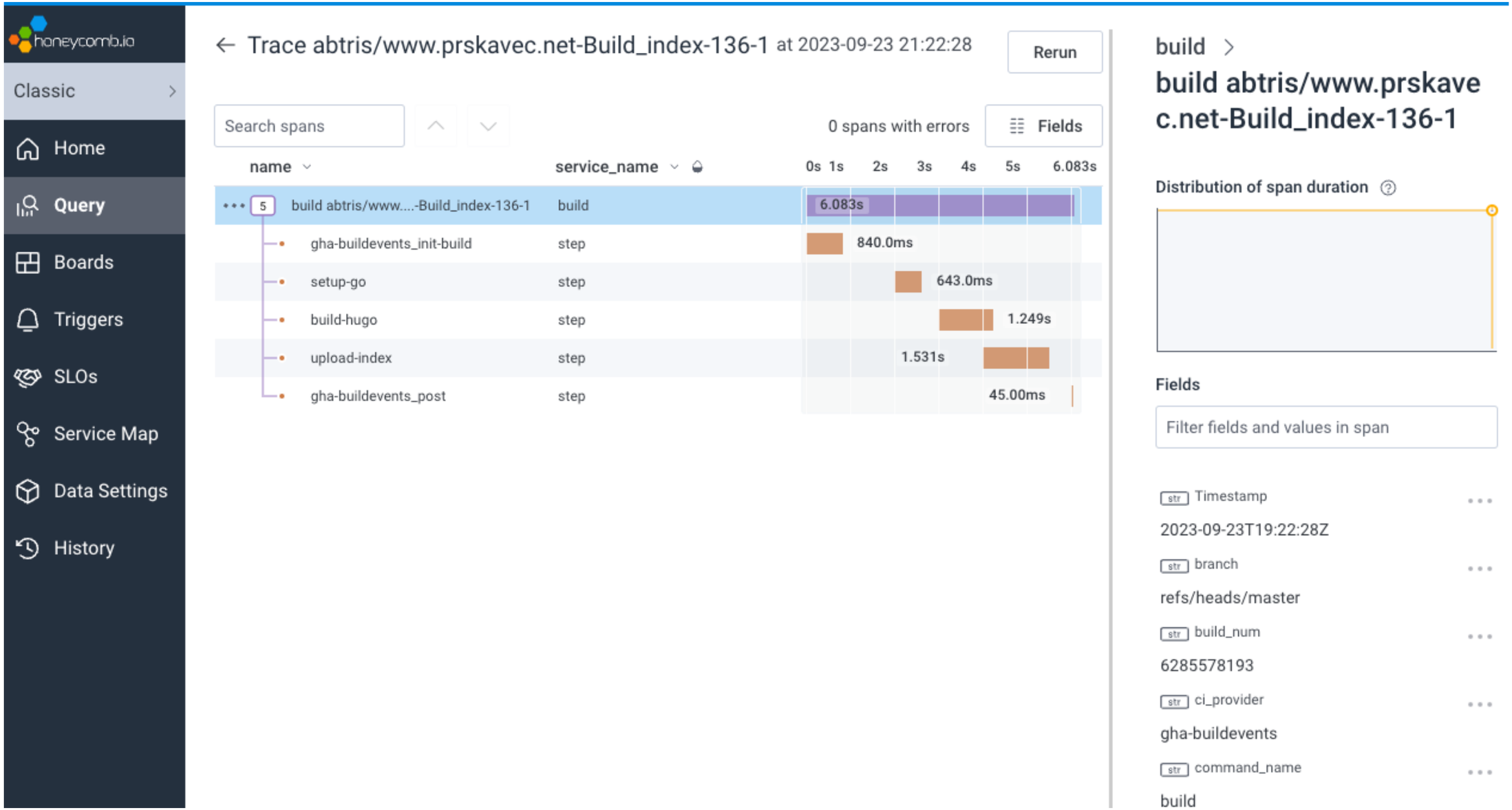
Configuration

```
export OTEL_EXPORTER_OTLP_ENDPOINT=api.honeycomb.io:443
export OTEL_EXPORTER_OTLP_HEADERS="x-honeycomb-team=your-api-key"
```

Honeycomb - buildevents

```yaml
...
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: honeycombio/gha-buildevents@v2
        with:
          apikey: ${{ secrets.BUILDEVENT_APIKEY }}
          dataset: gha-buildevents_integration
          status: ${{ job.status }}

      - name: Check out repo
        uses: actions/checkout@v3

      - name: Set up Go
        uses: actions/setup-go@v4
        with:
          go-version-file: 'go.mod'
      - run: |-
          STEP_ID=setup-go
          STEP_START=$(date +%s)

          go version

          buildevents step $TRACE_ID $STEP_ID $STEP_START $STEP_ID
...
```

→ GHA file - build.yml


honeycomb.io
Classic
Home
Query
Boards
Triggers
SLOs
Service Map
Data Settings
History
Trace abtris/www.prskavec.net-Build_index-136-1 at 2023-09-23 21:22:28
Rerun
Search spans
0 spans with errors
Fields
name
service_name
0s 1s 2s 3s 4s 5s 6.083s
5 build abtris/www....-Build_index-136-1 build 6.083s
gha-buildevents_init-build step 840.0ms
setup-go step 643.0ms
build-hugo step 1.249s
upload-index step 1.531s
gha-buildevents_post step 45.00ms
build
build abtris/www.prskavec.net-Build_index-136-1
Distribution of span duration
Fields
Filter fields and values in span
str Timestamp
2023-09-23T19:22:28Z
str branch
refs/heads/master
str build_num
6285578193
str ci_provider
gha-buildevents
str command_name
build

Tracking build steps

```
builtevents step $TRACE_ID $STEP_ID $STEP_START $STEP_NAME
```

- `$TRACE_ID` - Used to connect the current span to the larger workflow context. This variable is automatically set by the Honeycomb gha-buildevents Action.
- `$STEP_ID` - Used as a unique identifier for each span. It's just a string, but can often be handy to reference across steps, so it's best to store and use this as an environment variable.
- `$STEP_START` - A timestamp in the format of epoch time (seconds since 1970-1-1 UTC). It is used by buildevents to calculate the total duration of each step. You need to store this in an environment variable at invocation so it can be referenced when sending the output after the step completes.
- `$STEP_NAME` - A human-readable name for the step. This name will be used when displaying the trace view or when querying your trace data. It's possible to use the same value as `STEP_ID`, if the ID is set in a human-readable way.

Increasing granularity in your build steps

```
buildevents cmd $TRACE_ID $STEP_ID $CMD_NAME -- $CMD
```

- `$TRACE_ID` and `$STEP_ID` are unchanged.
  being both human-readable and a unique identifier.
- `$STEP_NAME` is substituted by `$CMD_NAME`. It maintains the same expectation of
- `$CMD` is added after two dashes and it contains any literal command to execute.
  By default, the command will be executed via `/bin/bash -c`.
- full example from honeycomb

Capturing events from builds that fail

- by default step that fail doesn't occur in traces
- decoupling the build step from the observability step creates two challenges.
- First, the default behavior of GitHub Actions is to stop the remaining workflow on failure.

```
- id: "Honeycomb: Finalize pact-credit-score"
  if: always()
  run: buildevents step $TRACE_ID $STEP_ID $STEP_START $STEP_ID
```

- In the event of a failed build span, this "always" case will run every finalize step, even those that were skipped due to a previous failure. Therefore, you need to add one more check within each finalize job to only send a new step to Honeycomb when the dependent job has run.
- To achieve this, the following code will always run the finalize step, but it uses the step context to only send the span if the relevant work step has run to completion

```
- id: "Honeycomb: Finalize pact-credit-score"
  if: always()
  env:
    OUTCOME: ${{ steps.pact-credit-score.outcome }}
  run: |
    if echo $OUTCOME | grep -wq -e success -e failure; then
      buildevents step $TRACE_ID $STEP_ID $STEP_START $STEP_ID
  fi
```

Do the following to each of the build and pact steps in your workflow:

- Move each invocation of buildevents step to its own finalize span step.
- Ensure it runs along with each prior service build step by using if: `always()`.
- Use the outcome of the working step to conditionally send a finalize step.
- Append variables in the service build steps to `$GITHUB_ENV`.

```
- id: build-special-membership
  run: |
    echo "STEP_ID=build_special-membership" >> $GITHUB_ENV
    echo "STEP_START=$(date +%s)" >> $GITHUB_ENV
    source $GITHUB_ENV

    # build
    buildevents cmd $TRACE_ID $STEP_ID 'build' -- \
      mvn clean verify -pl special-membership-service -Pcode-coverage -Pstatic-code-analysis

- name: "Honeycomb: Finalize build-special-membership"
  if: always()
  env:
    OUTCOME: ${{ steps.build-special-membership.outcome }}
  run: |
    if echo $OUTCOME | grep -wq -e success -e failure; then
      buildevents step $TRACE_ID $STEP_ID $STEP_START $STEP_ID
    fi
```

OTEL CLI

OTEL CLI

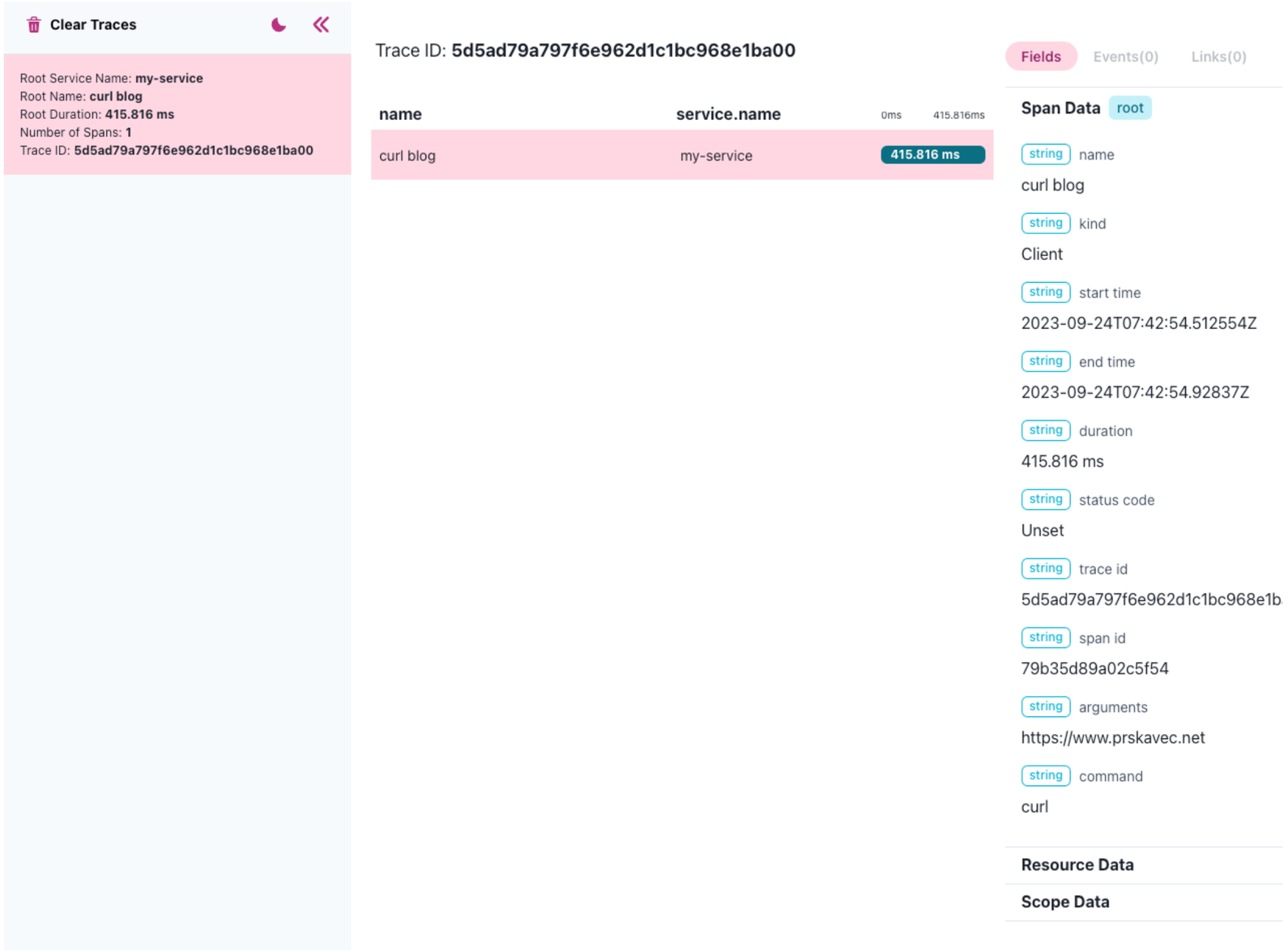
- easy local development using `otel-cli server tui` or `otel-desktop-viewer`
- support multiple vendors (Honeycomb, Lightstep, Elastic)

```
export LIGHTSTEP_TOKEN= # Lightstep API key (otlp/1 in the yaml)
export HONEYCOMB_TEAM=  # Honeycomb API key (otlp/2 in the yaml)
export HONEYCOMB_DATASET=playground # Honeycomb dataset
export ELASTIC_TOKEN= # Elastic token for the APM server.
```

- run a program inside a span
- propagates context via envvars so you can chain it to create child spans
- for advanced cases you can start a span in the background, and add events to it, finally closing it later in your script

Simple example

```
otel-desktop-viewer

# configure otel-cli to send to our desktop viewer endpoint
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4318

# use otel-cli to generate spans!
otel-cli exec --service my-service --name "curl blog" curl https://www.prskavec.net
```


Clear Traces
Root Service Name: my-service
Root Name: curl blog
Root Duration: 415.816 ms
Number of Spans: 1
Trace ID: 5d5ad79a797f6e962d1c1bc968e1ba00
Trace ID: 5d5ad79a797f6e962d1c1bc968e1ba00
name
service.name
0ms
415.816ms
curl blog
my-service
415.816 ms
Fields
Events(0)
Links(0)
Span Data
root
string
name
curl blog
string
kind
Client
string
start time
2023-09-24T07:42:54.512554Z
string
end time
2023-09-24T07:42:54.92837Z
string
duration
415.816 ms
string
status code
Unset
string
trace id
5d5ad79a797f6e962d1c1bc968e1b
string
span id
79b35d89a02c5f54
string
arguments
https://www.prskavec.net
string
command
curl
Resource Data
Scope Data

Instrumenting my pipeline in Dagger

OTEL Setup in your code

```
import (
  ...
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/codes"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracehttp"
    "go.opentelemetry.io/otel/sdk/resource"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/semconv/v1.21.0"
    "go.opentelemetry.io/otel/trace"
  ...
)
```

```
const (
    instrumentationName    = "github.com/abtris/dagger-tutorial"
    instrumentationVersion = "0.1.0"
)

var (
    tracer = otel.GetTracerProvider().Tracer(
        instrumentationName,
        trace.WithInstrumentationVersion(instrumentationVersion),
        trace.WithSchemaURL(semconv.SchemaURL),
    )
    sc trace.SpanContext
)
```

```go
ctx := context.Background()
opts := otlptracehttp.WithInsecure()
client := otlptracehttp.NewClient(opts)
exporter, err := otlptrace.New(ctx, client)
if err != nil {
  fmt.Errorf("creating OTLP trace exporter: %w", err)
}
tracerProvider := sdktrace.NewTracerProvider(
  sdktrace.WithBatcher(exporter),
  sdktrace.WithResource(resource.Default()),
)
otel.SetTracerProvider(tracerProvider)

// Handle shutdown properly so nothing leaks.
defer func() { _ = tracerProvider.Shutdown(ctx) }()
```

Manual Instrumentation

- Into span you can add events, attributes, errors

```
func build(ctx context.Context, repoUrl string) error {
    ctx, span = tracer.Start(ctx, "initDagger")
    span.AddEvent("start init dagger")
    span.SetAttributes(attribute.Bool("cache", true))
    client, err := dagger.Connect(ctx, dagger.WithLogOutput(os.Stdout))
    if err != nil {
        span.SetStatus(codes.Error, err.Error())
        span.RecordError(err)
        return err
```

- don't forget close the span `span.End()` or better

```
defer span.End()
```

Demo

→ setup local viewer for traces

```
export OTEL_EXPORTER_OTLP_ENDPOINT="http://localhost:4318"
export OTEL_TRACES_EXPORTER="otlp"
export OTEL_EXPORTER_OTLP_PROTOCOL="http/protobuf"
otel-desktop-viewer
```

→ source code

Questions?

Follow me on @abtris@hachyderm.io