Rust

The OpenTelemetry Rust project has several examples for tracing.

There are several crates worth mentioning for Rust developers:

- opentelemetry_jaeger was created to send traces using Jaeger Propagation format. There are many tutorials that still rely on this crate. However, bear in mind that this whole crate is deprecated and should no longer be used.
- opentelemetry_otlp is the official crate to build an Exporter.
- tracing_opentelemetry is required if you want to integrate OpenObserve with Tokio's tracing project.

Example with the Tracing framework

This example is based on the one provided by tracing_opentelemetry, with all the necessary configuration to make it work with OpenObserve.

```
use std::io::Error;

use opentelemetry::{global, trace::TracerProvider, Key, KeyValue};
use opentelemetry_otlp::WithExportConfig;
use opentelemetry_sdk::{
    metrics::{
        reader::{DefaultAggregationSelector, DefaultTemporalitySelector},
        Aggregation, Instrument, MeterProviderBuilder, PeriodicReader,
SdkMeterProvider, Stream,
    },
    runtime,
    trace::{BatchConfig, RandomIdGenerator, Sampler, Tracer},
    Resource,
};
use opentelemetry_semantic_conventions::{
    resource::{DEPLOYMENT_ENVIRONMENT, SERVICE_NAME, SERVICE_VERSION},
    SCHEMA_URL,
};
use tonic::metadata::*;
use tracing_core::Level;
use tracing_opentelemetry::{MetricsLayer, OpenTelemetryLayer};
use tracing_subscriber::{layer::SubscriberExt, util::SubscriberInitExt};

// Create a Resource that captures information about the entity for which
```

```
telemetry is recorded.
fn resource() -> Resource {
    Resource::from_schema_url(
        [
            KeyValue::new(SERVICE_NAME, env!("CARGO_PKG_NAME")),
            KeyValue::new(SERVICE_VERSION, env!("CARGO_PKG_VERSION")),
            KeyValue::new(DEPLOYMENT_ENVIRONMENT, "develop"),
        ],
        SCHEMA_URL,
    )
}

// Create the required Metadata headers for OpenObserve
fn otl_metadata() -> Result<MetadataMap, Error> {
    let mut map = MetadataMap::with_capacity(3);

    map.insert(
        "authorization",
        format!("Basic cm9vdEBleGFtcGxlLmNvbTo3a2xCT052cHhTd09DZ09u") // This is
picked from the Ingestion tab openobserve
            .parse()
            .unwrap(),
    );
    map.insert("organization", "default".parse().unwrap());
    map.insert("stream-name", "default".parse().unwrap());
    Ok(map)
}

// Construct MeterProvider for MetricsLayer
fn init_meter_provider() -> SdkMeterProvider {
    let exporter = opentelemetry_otlp::new_exporter()
        .tonic()
        .with_endpoint("http://localhost:5081/api/development")
        .with_protocol(opentelemetry_otlp::Protocol::Grpc)
        .with_metadata(otl_metadata().unwrap())
        .build_metrics_exporter(
            Box::new(DefaultAggregationSelector::new()),
            Box::new(DefaultTemporalitySelector::new()),
        )
        .unwrap();

    let reader = PeriodicReader::builder(exporter, runtime::Tokio)
        .with_interval(std::time::Duration::from_secs(30))
        .build();

    // For debugging in development
    let stdout_reader = PeriodicReader::builder(
        opentelemetry_stdout::MetricsExporter::default(),
        runtime::Tokio,
    )
    .build();

    // Rename foo metrics to foo_named and drop key_2 attribute
```

```
    let view_foo = |instrument: &Instrument| -> Option<Stream> {
        if instrument.name == "foo" {
            Some(
                Stream::new()
                    .name("foo_named")
                    .allowed_attribute_keys([Key::from("key_1")]),
            )
        } else {
            None
        }
    };

    // Set Custom histogram boundaries for baz metrics
    let view_baz = |instrument: &Instrument| -> Option<Stream> {
        if instrument.name == "baz" {
            Some(
                Stream::new()
                    .name("baz")
                    .aggregation(Aggregation::ExplicitBucketHistogram {
                        boundaries: vec![0.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0],
                        record_min_max: true,
                    }),
            )
        } else {
            None
        }
    };

    let meter_provider = MeterProviderBuilder::default()
        .with_resource(resource())
        .with_reader(reader)
        .with_reader(stdout_reader)
        .with_view(view_foo)
        .with_view(view_baz)
        .build();

    global::set_meter_provider(meter_provider.clone());

    meter_provider
}

// Construct Tracer for OpenTelemetryLayer
fn init_tracer() -> Tracer {
    let provider = opentelemetry_otlp::new_pipeline()
        .tracing()
        .with_trace_config(
            opentelemetry_sdk::trace::Config::default()
                // Customize sampling strategy

.with_sampler(Sampler::ParentBased(Box::new(Sampler::TraceIdRatioBased(
                    1.0,
                ))))
                // If export trace to AWS X-Ray, you can use XrayIdGenerator
```

```
                .with_id_generator(RandomIdGenerator::default())
                .with_resource(resource()),
        )
        .with_batch_config(BatchConfig::default())
        .with_exporter(
            opentelemetry_otlp::new_exporter()
                .tonic()
                .with_endpoint("http://localhost:5081/api/development")
                .with_metadata(otl_metadata().unwrap()),
        )
        .install_batch(runtime::Tokio)
        .unwrap();

    global::set_tracer_provider(provider.clone());
    provider.tracer("tracing-otel-subscriber")
}

// Initialize tracing-subscriber and return OtelGuard for opentelemetry-related
termination processing
fn init_tracing_subscriber() -> OtelGuard {
    let meter_provider = init_meter_provider();
    let tracer = init_tracer();

    tracing_subscriber::registry()
        .with(tracing_subscriber::filter::LevelFilter::from_level(
            Level::INFO,
        ))
        .with(tracing_subscriber::fmt::layer())
        .with(MetricsLayer::new(meter_provider.clone()))
        .with(OpenTelemetryLayer::new(tracer))
        .init();

    OtelGuard { meter_provider }
}

struct OtelGuard {
    meter_provider: SdkMeterProvider,
}

impl Drop for OtelGuard {
    fn drop(&mut self) {
        if let Err(err) = self.meter_provider.shutdown() {
            eprintln!("{err:?}");
        }
        opentelemetry::global::shutdown_tracer_provider();
    }
}

#[tokio::main]
async fn main() {
    let _guard = init_tracing_subscriber();

    foo().await;
```

```
}

#[tracing::instrument]
async fn foo() {
    tracing::info!(
        monotonic_counter.foo = 1_u64,
        key_1 = "bar",
        key_2 = 10,
        "handle foo",
    );

    tracing::info!(histogram.baz = 10, "histogram example",);
}
```

Dependencies required by the above example

```
opentelemetry = "0.24.0"
opentelemetry-otlp = "0.17.0"
opentelemetry-semantic-conventions = "0.16.0"
opentelemetry-stdout = "0.5.0"
opentelemetry_sdk = { version = "0.24.1", features = ["rt-tokio"] }
tokio = { version = "1.39.2", features = ["full"] }
tonic = "0.12.1"
tracing = "0.1.40"
tracing-core = "0.1.32"
tracing-opentelemetry = "0.25.0"
tracing-subscriber = { version = "0.3.18" }
```