Task: Implement Rust Tracing with Elastic OpenTelemetry

Objective: To integrate distributed tracing into our Rust applications using the tracing framework and export the telemetry data to our Elastic Stack via OpenTelemetry. This will enhance our ability to monitor, debug, and understand the performance of our Rust services in a production environment.

1. Understanding the Integration Architecture

Before starting, familiarize yourself with the key components involved in this integration:

- **tracing**: Our primary Rust instrumentation framework for structured logging and distributed tracing. It allows us to define spans (operations) and events (points in time).
- **tracing-opentelemetry**: This crate acts as the bridge, translating tracing's internal data model into OpenTelemetry-compatible traces and events.
- **opentelemetry & opentelemetry-sdk**: The core OpenTelemetry Rust libraries for generating, processing, and managing telemetry data.
- **opentelemetry-otlp**: The OpenTelemetry Protocol (OTLP) exporter, responsible for sending the telemetry data. We will use OTLP over HTTP/protobuf.
- **OpenTelemetry Collector**: A crucial intermediary service that receives telemetry data from our applications, processes it, and forwards it to the Elastic Stack. This provides buffering, processing, and resilience.
- **Elastic Stack (APM Server, Elasticsearch, Kibana)**: Our centralized observability platform where traces will be ingested (APM Server), stored (Elasticsearch), and visualized/analyzed (Kibana).

2. Prerequisites

Before beginning the implementation, ensure the following are in place:

- **Rust Toolchain**: Latest stable Rust and Cargo installed.
- **OpenTelemetry Collector**: The opentelemetry-collector-contrib executable downloaded and accessible.
- **Elastic Stack**: Access to a running Elastic APM Server, Elasticsearch, and Kibana instance. You will need the APM Server's OTLP HTTP endpoint (typically port 8200).
- **Basic Understanding**: Familiarity with Rust's asynchronous programming (if applicable) and the concepts of distributed tracing (spans, events, trace context).

3. Implementation Steps

Follow these steps to integrate tracing with Elastic OpenTelemetry in your Rust application.

Task 3.1: Update Cargo.toml Dependencies

Add the following dependencies to your project's Cargo.toml. Pay attention to version compatibility, especially between tracing-opentelemetry and the opentelemetry-* crates.
[dependencies]
# Core tracing library for instrumentation
tracing = "0.1"

Subscriber for tracing, enabling environment filter and formatted output
tracing-subscriber = { version = "0.3", features = ["env-filter", "fmt"] }

Bridge between tracing and OpenTelemetry
IMPORTANT: Check compatibility with your opentelemetry-* versions.
tracing-opentelemetry = "0.23"

# Core OpenTelemetry API and SDK
opentelemetry = { version = "0.22", features = ["trace", "metrics"] } # Enable "trace" feature for tracing
opentelemetry-sdk = { version = "0.22", features = ["trace", "rt-tokio"] } # Enable "trace" and a runtime feature (e.g., "rt-tokio")

OpenTelemetry Protocol (OTLP) exporter for sending data
"http-proto" for HTTP/protobuf, "reqwest-client" for the HTTP client, "tokio" for async runtime integration
opentelemetry-otlp = { version = "0.15", features = ["trace", "http-proto", "reqwest-client", "tokio"] }

# Optional: For asynchronous runtime (e.g., if you're building a web service)
tokio = { version = "1", features = ["full"] }

Optional: For standard OpenTelemetry semantic conventions (e.g., "service.name")
opentelemetry-semantic-conventions = "0.14"

Task 3.2: Initialize OpenTelemetry and tracing Subscriber

Implement the following initialization logic at the entry point of your application (e.g., in main.rs). This code sets up the OpenTelemetry TracerProvider and configures the tracing subscriber to forward spans and events to OpenTelemetry.
Action: Copy and adapt the following code snippet into your application's main.rs (or equivalent).
use opentelemetry::{
  global,
  sdk::{
    trace::{self, Sampler},
    Resource,

```
    },
    KeyValue,
};
use opentelemetry_otlp::{self, WithExportConfig};
use tracing::{info, instrument, Span};
use tracing_subscriber::{prelude::*, registry::Registry, EnvFilter};

// Use #[tokio::main] if your application is asynchronous and uses Tokio
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // 1. Configure OpenTelemetry Tracer Provider
    // This sets up the OTLP HTTP/protobuf exporter.
    // The `with_endpoint` should point to your OpenTelemetry Collector or Elastic APM Server.
    let otlp_exporter = opentelemetry_otlp::new_exporter()
        .http()
        .with_endpoint("http://localhost:4318/v1/traces") // Default OTLP HTTP endpoint for
Collector
        .with_timeout(std::time::Duration::from_secs(3)) // Set a timeout for export operations
        .build()?;

    // Create the OpenTelemetry tracer pipeline
    let tracer = opentelemetry_otlp::new_pipeline()
        .tracing() // Specify that this pipeline is for tracing
        .with_exporter(otlp_exporter) // Attach our OTLP exporter
        .with_trace_config(
            trace::config()
                .with_sampler(Sampler::AlwaysOn) // Configure sampling: AlwaysOn means all traces
are recorded
                .with_resource(Resource::new(vec![
                    // Define resource attributes for your service.
                    // These attributes describe the entity producing the telemetry.
                    KeyValue::new("service.name", "my-rust-app"),
                    KeyValue::new("service.version", "0.1.0"),
                    KeyValue::new("host.name", "my-server-123"),
                    // Add more attributes as needed (e.g., deployment.environment,
service.instance.id)
                ])),
        )
        // Install the batch tracer, using the Tokio runtime for asynchronous operations
        .install_batch(opentelemetry::runtime::Tokio)?;

    // 2. Create a tracing-opentelemetry layer
    // This layer bridges tracing spans/events to OpenTelemetry spans/events.
```

```
    let opentelemetry_layer = tracing_opentelemetry::OpenTelemetryLayer::new(tracer);

    // 3. Create a tracing subscriber
    // A subscriber combines multiple layers to process tracing data.
    // - `EnvFilter`: Allows controlling log levels via the RUST_LOG environment variable (e.g.,
RUST_LOG=info,my_app=debug).
    // - `opentelemetry_layer`: Sends traces to OpenTelemetry.
    // - `tracing_subscriber::fmt::layer().compact()`: Provides formatted console output, useful
for local development.
    let subscriber = Registry::default()
        .with(EnvFilter::from_default_env())
        .with(opentelemetry_layer)
        .with(tracing_subscriber::fmt::layer().compact());

    // 4. Set the subscriber as the global default
    // Once set, all tracing macros will use this subscriber.
    global::set_subscriber(subscriber)?;

    // --- Your application logic starts here ---
    info!("Application started, performing operations...");

    // Call instrumented functions
    my_instrumented_function(10, 20).await;
    another_function().await;

    info!("Application finished.");

    // Ensure all buffered spans are flushed before the application exits.
    // This is crucial to prevent data loss, especially in short-lived applications.
    global::shutdown_tracer_provider();

    Ok(())
}

/// An example function instrumented with `#[instrument]`.
/// This macro automatically creates a span for the function call,
/// capturing its name and arguments as span attributes.
#[instrument]
async fn my_instrumented_function(a: u32, b: u32) -> u32 {
    info!("Inside my_instrumented_function: Adding two numbers.");
    // An event within the current span
    tracing::event!(tracing::Level::DEBUG, "Performing the addition operation.");
    let result = a + b;
```

```
    // Manually create a nested span for a specific part of the logic.
    // The `result` variable is added as an attribute to this inner span.
    let inner_span = tracing::span!(tracing::Level::INFO, "inner_calculation", result = result);
    // Enter the span. The `_guard` ensures the span is exited when it goes out of scope.
    let _guard = inner_span.enter();
    info!("Inner calculation logic completed.");
    // The span automatically ends when `_guard` is dropped.
    drop(_guard); // Explicitly dropping for clarity, not strictly necessary here.

    info!("Exiting my_instrumented_function.");
    result
}

/// Another example function, demonstrating custom fields and span attributes.
#[instrument(fields(custom_field = "initial_value"))] // Add a default field to the span
async fn another_function() {
    info!("Executing another_function.");
    // Get the current span and record additional attributes dynamically.
    Span::current().record("step", "initialized_async_task");

    // Simulate some asynchronous work
    tokio::time::sleep(tokio::time::Duration::from_millis(50)).await;

    // Update an existing attribute or add a new one
    Span::current().record("step", "completed_async_task");
    Span::current().record("status", "success");

    info!("another_function done.");
}
```

Task 3.3: Configure and Run OpenTelemetry Collector

The OpenTelemetry Collector will receive traces from your Rust application and forward them to Elastic APM.

Action:

1. **Download**: If you haven't already, download the opentelemetry-collector-contrib executable for your operating system from the official OpenTelemetry GitHub releases page.
2. **Configuration File**: Create a file named otel-collector-config.yaml with the following content.
3. **Update Endpoint**: **Crucially, replace <your-elastic-apm-server-host>:8200 with the actual endpoint of our Elastic APM Server.**

```
# otel-collector-config.yaml
receivers:
  otlp:
    protocols:
      grpc: # Enable gRPC receiver
      http: # Enable HTTP receiver
        endpoint: 0.0.0.0:4318 # Default OTLP HTTP port. Your Rust app sends data here.

processors:
  batch: # Batch spans for efficient export to the backend
    send_batch_size: 100
    timeout: 1s
  resource: # Add common resource attributes to all telemetry data
    attributes:
      - key: host.name
        value: my-rust-collector-host # Identify the host running the collector
        action: upsert

exporters:
  otlphttp/elastic: # Define an OTLP HTTP exporter specifically for Elastic
    endpoint: "http://<your-elastic-apm-server-host>:8200" # **IMPORTANT: Replace with your
Elastic APM Server URL**
    compression: gzip # Recommended for production for better performance
    headers:
      # Uncomment and configure if your Elastic APM Server requires authentication
      # authorization: "Bearer <your_elastic_apm_api_key>"
      # For Basic Auth, base64 encode "username:password" like this: echo -n
"username:password" | base64
      # authorization: "Basic <base64_encoded_credentials>"

service:
  pipelines:
    traces: # Define the trace processing pipeline
      receivers: [otlp] # Receive traces from the OTLP receiver
      processors: [batch, resource] # Process traces with batching and resource enrichment
      exporters: [otlphttp/elastic] # Export processed traces to Elastic APM
    # You can add similar pipelines for metrics and logs if you collect them:
    # metrics:
    #   receivers: [otlp]
    #   processors: [batch, resource]
    #   exporters: [otlphttp/elastic]
    # logs:
    #   receivers: [otlp]
```

```
  # processors: [batch, resource]
  # exporters: [otlphttp/elastic]
```

Action: Run the collector from your terminal, in the same directory as the configuration file:
./opentelemetry-collector-contrib --config otel-collector-config.yaml

Task 3.4: Run Your Rust Application

With the OpenTelemetry Collector running, compile and execute your Rust application.
Action: In a separate terminal, navigate to your Rust project directory and run:
cargo run

Your Rust application will now send its tracing data, transformed into OpenTelemetry traces, to the OpenTelemetry Collector, which will then forward them to our Elastic APM Server.

4. Verification and Deliverables

Deliverable: Confirmation that traces from your Rust application are successfully appearing in Kibana.
Verification Steps:

1. **Open Kibana**: Access our Kibana instance via your web browser.
2. Navigate to **Observability** -> **APM**.
3. Look for your my-rust-app service listed. Click on it.
4. Verify that transactions and traces are appearing. Explore individual traces to confirm that the spans (my_instrumented_function, inner_calculation, another_function), their durations, events, and attributes are correctly displayed.

5. Key Concepts and Best Practices (Tips for Success)

Keep these points in mind during and after implementation:

- **#[instrument] Macro**: Leverage this macro for automatic span creation on functions. It's concise and automatically captures function arguments as span fields.
- **tracing::span! Macro**: Use this for fine-grained control, creating spans for specific logical operations or code blocks within a function.
- **tracing::event! Macro**: For capturing discrete, point-in-time occurrences. These are excellent for structured logging within a span's context, providing valuable details about an operation.
- **Context Propagation**: For distributed tracing across multiple services, ensure trace context (Trace ID, Span ID) is correctly propagated. OpenTelemetry handles this automatically for many common protocols.
- **Resource Attributes**: Define meaningful service.name, service.version, host.name, deployment.environment, etc. These are crucial for organizing and filtering telemetry data in Kibana.
- **Sampling**: In production, consider implementing sampling strategies (e.g.,

Sampler::TraceIdRatioBased) to manage the volume of telemetry data.

- **Asynchronous Runtimes**: If your application is asynchronous, confirm that the correct runtime features are enabled in your opentelemetry-sdk and opentelemetry-otlp dependencies (e.g., rt-tokio).
- **Error Handling and Shutdown**: Implement robust error handling for OpenTelemetry initialization and ensure global::shutdown_tracer_provider() is called before your application exits to flush any buffered telemetry data.
- **Structured Logging**: tracing naturally promotes structured logging, which makes your logs highly queryable and analyzable in Elasticsearch when correlated with traces.

This task provides the necessary steps to get our Rust applications integrated with our observability stack. Please reach out if you encounter any issues or have questions during the implementation.