**Rust multi module microservices Part 7 — Books Analytics**

[[](https://medium.com/@omprakashsridharan?source=post_page-----eeb3008db649--------------------------------)](https://medium.com/@omprakashsridharan?source=post_page-----eeb3008db649--------------------------------)

[Omprakash Sridharan](https://medium.com/@omprakashsridharan?source=post_page-----eeb3008db649--------------------------------)

·

Follow

3 min read

·

Jun 27

We are so close to the end and just one more step to configure a service that consumes the created book. Update the Cargo.toml of books\_analytics crate to have the below contents.

[package]  
name = "books\_analytics"  
version = "0.0.0"  
edition = "2021"  
  
# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html  
  
[dependencies]  
kafka = {path = "../kafka"}  
tokio = { workspace = true }  
common = {path = "../common"}  
tracing = {workspace = true}  
tracing-subscriber = {workspace = true}  
opentelemetry = {workspace = true}  
axum-tracing-opentelemetry = {workspace = true}  
opentelemetry-zipkin = {workspace = true}  
tracing-opentelemetry = {workspace = true}

Update the [main.rs](http://main.rs/) to have the below code

use common::events::{constants::Topics, dto::CreatedBook};  
use kafka::consumer::KafkaConsumer;  
use tokio::sync::mpsc;  
use tracing::info;  
use tracing\_subscriber::{layer::SubscriberExt, util::SubscriberInitExt, EnvFilter};  
  
#[tokio::main]  
async fn main() -> Result<(), Box<dyn std::error::Error>> {  
 opentelemetry::global::set\_text\_map\_propagator(opentelemetry\_zipkin::Propagator::new());  
 let tracer = opentelemetry\_zipkin::new\_pipeline()  
 .with\_service\_name("books\_analytics".to\_owned())  
 .with\_service\_address("127.0.0.1:8080".parse().unwrap())  
 .with\_collector\_endpoint("http://localhost:9411/api/v2/spans")  
 .install\_batch(opentelemetry::runtime::Tokio)  
 .expect("unable to install zipkin tracer");  
 let tracer = tracing\_opentelemetry::layer().with\_tracer(tracer.clone());  
  
 let subscriber = tracing\_subscriber::fmt::layer().json();  
  
 let level = EnvFilter::new("debug".to\_owned());  
  
 tracing\_subscriber::registry()  
 .with(subscriber)  
 .with(level)  
 .with(tracer)  
 .init();  
 let kakfa\_consumer = KafkaConsumer::new(  
 "localhost:9092".to\_string(),  
 "http://localhost:8081".to\_string(),  
 "books-created-consumer".to\_string(),  
 Topics::BookCreated.to\_string(),  
 );  
  
 let (sender, mut receiver) = mpsc::unbounded\_channel::<CreatedBook>();  
 tokio::spawn(async move {  
 info!("Strarting book created consumer");  
 kakfa\_consumer.consume(sender.clone()).await;  
 });  
  
 while let Some(message) = receiver.recv().await {  
 info!("Consumed messaged {:?}", message)  
 }  
 opentelemetry::global::shutdown\_tracer\_provider();  
 Ok(())  
}

1. The code starts by importing various modules and dependencies.
2. The main function is defined as an asynchronous function and is marked with the #[tokio::main] attribute, indicating that it should be executed as the entry point of the program.
3. The first line in the main function sets the text map propagator for OpenTelemetry to the Zipkin propagator.
4. The code then initializes a Zipkin tracer pipeline by configuring the service name, service address, and collector endpoint. The pipeline is installed and assigned to the tracer variable.
5. Next, the tracer is used to create a tracing layer (tracing\_opentelemetry::layer()) that is added to the subscriber.
6. The subscriber is configured with a JSON formatter and an environment filter to determine the log level.
7. The subscriber is then initialized using tracing\_subscriber::registry().init().
8. An instance of KafkaConsumer is created, passing the Kafka bootstrap servers, schema registry URL, consumer group ID, and the topic name as arguments.
9. An unbounded channel is created using tokio::sync::mpsc::unbounded\_channel, which returns a sender and a receiver.
10. A background task is spawned using tokio::spawn, which will consume messages from Kafka using the consume method of KafkaConsumer. The sender is cloned and passed as an argument to the consume method.
11. Inside the spawned task, a log message is printed indicating that the consumer is starting.
12. The while loop iterates over messages received from the receiver channel using receiver.recv().await. When a message is received, a log message is printed indicating that the message has been consumed.
13. After the loop exits, the global tracer provider is shut down.
14. Finally, the Ok(()) expression is returned to indicate a successful execution of the main function.

In summary, the code sets up the necessary components for consuming messages from a Kafka topic. It initializes tracing, creates a Kafka consumer, spawns a task to consume messages, and prints log messages for each consumed message. The program runs indefinitely until it is terminated.

We are done with the coding and now let’s run the services to see what all this is about in the next article.