**Rust multi module microservices Part 6 — Books API**

[[Omprakash Sridharan](https://medium.com/@omprakashsridharan?source=post_page-----c5b0ebed52da--------------------------------)](https://medium.com/@omprakashsridharan?source=post_page-----c5b0ebed52da--------------------------------)

[Omprakash Sridharan](https://medium.com/@omprakashsridharan?source=post_page-----c5b0ebed52da--------------------------------)

·

Follow

13 min read

·

Jun 27

3

1

I hope you’ve reached this point, and now we’ll begin creating our first microservice application: a books HTTP REST API with a single POST endpoint for adding a book. However, before we proceed, we must establish a migration setup for this service to store the book data in a database.

**Migration**

Let us install the sea-orm cli which has instructions to setup migration[here](https://www.sea-ql.org/SeaORM/docs/migration/setting-up-migration/).

cargo install sea-orm-cli

We will go into the books\_api directory and initialise a migration

cd books\_api && sea-orm-cli migrate init

After that, we will have a structure like below

migration  
├── Cargo.toml  
├── README.md  
└── src  
 ├── lib.rs # Migrator API, for integration  
 ├── m20220101\_000001\_create\_table.rs # A sample migration file  
 └── main.rs # Migrator CLI, for running manually

Update the Cargo.toml inside the newly created migration directory to have the following content.

[package]  
name = "migration"  
version = "0.1.0"  
edition = "2021"  
publish = false  
  
[lib]  
name = "migration"  
path = "src/lib.rs"  
  
[dependencies]  
tokio = { workspace = true }  
[dependencies.sea-orm-migration]  
version = "^0"  
features = [  
 "runtime-tokio-rustls", "sqlx-postgres"  
]

By default, an initial create table migration file is setup for us and let us update it to create a book table. You can do much more and please follow the sea-orm documentation of the possibilities like seeding with initial data, enums, etc. I am just scratching the top surface here.

use sea\_orm\_migration::prelude::\*;  
  
#[derive(DeriveMigrationName)]  
pub struct Migration;  
  
#[async\_trait::async\_trait]  
impl MigrationTrait for Migration {  
 async fn up(&self, manager: &SchemaManager) -> Result<(), DbErr> {  
 manager  
 .create\_table(  
 Table::create()  
 .table(Book::Table)  
 .if\_not\_exists()  
 .col(  
 ColumnDef::new(Book::Id)  
 .integer()  
 .not\_null()  
 .auto\_increment()  
 .primary\_key(),  
 )  
 .col(ColumnDef::new(Book::Title).string().unique\_key().not\_null())  
 .col(ColumnDef::new(Book::Isbn).string().not\_null())  
 .to\_owned(),  
 )  
 .await  
 }  
  
 async fn down(&self, manager: &SchemaManager) -> Result<(), DbErr> {  
 manager  
 .drop\_table(Table::drop().table(Book::Table).to\_owned())  
 .await  
 }  
}  
  
/// Learn more at https://docs.rs/sea-query#iden  
#[derive(Iden)]  
enum Book {  
 Table,  
 Id,  
 Title,  
 Isbn,  
}

We can run the migrations by navigating to the migration directory and checkout the [README.md](http://readme.md/) that got generated for available commands.

**Terminal 1**

docker compose up

**Terminal 2**

cd books\_api/migration # Press enter  
DATABASE\_URL=postgres://postgres:postgres@localhost:5433/rust-superapp cargo run

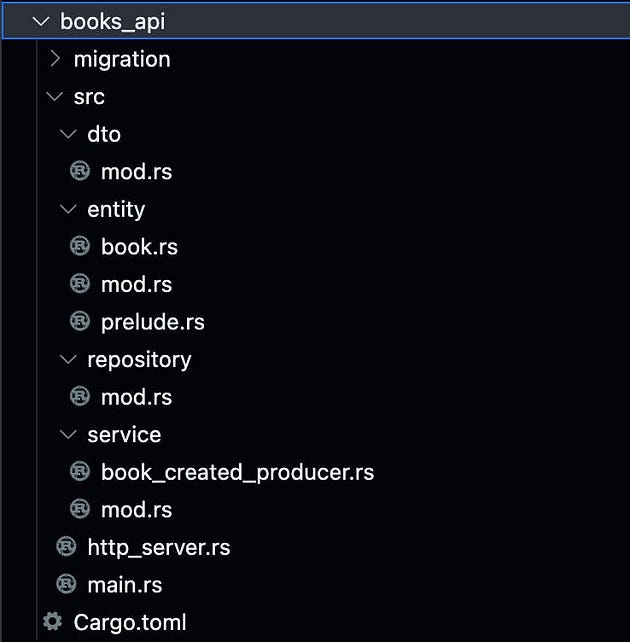
**books-api/Cargo.toml**

Let’s update the books\_api Cargo.toml to include all the necessary dependencies

[package]  
name = "books\_api"  
version = "0.0.0"  
edition = "2021"  
  
# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html  
  
[dependencies]  
migration = { path = "migration" }  
database = { path = "../database" }  
tokio = { workspace = true }  
tracing = { workspace = true }  
tracing-subscriber = { workspace = true }  
serde = { workspace = true }  
sea-orm = { workspace = true }  
derive\_builder = { workspace = true }  
thiserror = { workspace = true }  
testcontainers = { workspace = true }  
kafka = {path = "../kafka"}  
common = {path = "../common"}  
axum = {workspace = true}  
serde\_json = {workspace = true}  
opentelemetry = {workspace = true}  
axum-tracing-opentelemetry = {workspace = true}  
opentelemetry-zipkin = {workspace = true}  
tracing-opentelemetry = {workspace = true}  
schema\_registry\_converter = {workspace = true}  
apache-avro = {workspace = true}

Aside from the ‘workspace = true’ dependencies, we have now added four more path-based dependencies. If you recall, we are currently sharing our common, Kafka, and database library crates. 🎉

**Structure**



Go ahead and create the necessary folders & files inside books\_api as you see above except the entity directory and we will start the implementation.

**Entity**

Let’s start by creating the entities that will be used to map with the database table for CRUD operations. However, the best part is that we don’t need to write them manually; instead, we can generate them. Run the following command to generate the entities. (Note: For this to work, the migrations mentioned above should have been run.)

cd books\_api # Make sure you are here inside books\_api directory  
sea-orm-cli generate entity -u postgres://postgres:postgres@localhost/rust-superapp --with-serde both -o ./src/entity

This command will generate the entity directory containing three files: [book.rs](http://book.rs/), [mod.rs](http://mod.rs/), and [prelude.rs](http://prelude.rs/). Upon inspection, the files should be self-explanatory. If you choose to use this auto-generation method, it is recommended not to edit the files manually, as any future generations will overwrite the manual changes. However, you can also create your own entities by following the SeaORM documentation provided [here](https://www.sea-ql.org/SeaORM/docs/generate-entity/entity-structure/).

**DTO**

Update the src/dto/[mod.rs](http://mod.rs/) file with the following code, which creates a domain object representing a book. This structure will be utilized for any business logic and will not be used for database or Kafka messages. This approach offers a solid abstraction between the domain and infrastructure, inherently allowing us to make future refactoring with fewer issues.

use derive\_builder::Builder;  
use serde::{Deserialize, Serialize};  
  
#[derive(Serialize, Deserialize, Builder, Debug)]  
pub struct Book {  
 pub id: i32,  
 pub title: String,  
 pub isbn: String,  
}

**Repository**

Update the src/repository/[mod.rs](http://mod.rs/) with the following code, which holds methods for saving a book to the database. Here we also run the migration whenever the repository is initialised so that this process automatically keeps our database state in sync with the application entity structure. If any error, the application startup will fail indicating a dirty state which can be rectified and redeployed.

use crate::entity::book::{ActiveModel as BookActiveModel, Model as BookModel};  
use migration::{Migrator, MigratorTrait};  
use sea\_orm::ActiveValue::Set;  
use sea\_orm::{ActiveModelTrait, DatabaseConnection, DbErr};  
use std::sync::Arc;  
use thiserror::Error;  
  
#[derive(Clone)]  
pub struct Repository {  
 database\_connection: Arc<DatabaseConnection>,  
}  
  
#[derive(Error, Debug)]  
pub enum RepositoryError {  
 #[error("Database error")]  
 DatabaseError(#[from] DbErr),  
}  
  
impl Repository {  
 pub async fn new(database\_connection: DatabaseConnection) -> Result<Self, RepositoryError> {  
 Migrator::up(&database\_connection, None)  
 .await  
 .map\_err(|e| RepositoryError::DatabaseError(e))?;  
 Ok(Self {  
 database\_connection: Arc::new(database\_connection),  
 })  
 }  
  
 pub async fn create\_book(  
 &self,  
 title: String,  
 isbn: String,  
 ) -> Result<BookModel, RepositoryError> {  
 let created\_book = BookActiveModel {  
 title: Set(title),  
 isbn: Set(isbn),  
 ..Default::default()  
 };  
 created\_book  
 .insert(self.database\_connection.as\_ref())  
 .await  
 .map\_err(|e| RepositoryError::DatabaseError(e))  
 }  
}  
  
#[cfg(test)]  
mod tests {  
 use crate::repository::Repository;  
 use database::get\_connection;  
 use testcontainers::{clients, images};  
  
 #[tokio::test]  
 async fn test\_create\_book() {  
 let docker = clients::Cli::default();  
 let database = images::postgres::Postgres::default();  
 let node = docker.run(database);  
 let connection\_string = &format!(  
 "postgres://postgres:postgres@127.0.0.1:{}/postgres",  
 node.get\_host\_port\_ipv4(5432)  
 );  
 let database\_connection = get\_connection(connection\_string).await.unwrap();  
 let repository = Repository::new(database\_connection.clone()).await.unwrap();  
 let title = "TITLE".to\_string();  
 let isbn = "ISBN".to\_string();  
 let created\_book = repository  
 .create\_book(title.clone(), isbn.clone())  
 .await  
 .unwrap();  
 assert\_eq!(created\_book.title, title.clone());  
 assert\_eq!(created\_book.isbn, isbn.clone());  
 }  
  
 #[tokio::test]  
 async fn test\_create\_book\_title\_unique() {  
 let docker = clients::Cli::default();  
 let database = images::postgres::Postgres::default();  
 let node = docker.run(database);  
 let connection\_string = &format!(  
 "postgres://postgres:postgres@127.0.0.1:{}/postgres",  
 node.get\_host\_port\_ipv4(5432)  
 );  
 let database\_connection = get\_connection(connection\_string).await.unwrap();  
 let repository = Repository::new(database\_connection.clone()).await.unwrap();  
 let title = "TITLE".to\_string();  
 let isbn = "ISBN".to\_string();  
 let created\_book1 = repository  
 .create\_book(title.clone(), isbn.clone())  
 .await  
 .unwrap();  
 assert\_eq!(created\_book1.title, title.clone());  
 assert\_eq!(created\_book1.isbn, isbn.clone());  
  
 let created\_book2\_result = repository.create\_book(title.clone(), isbn.clone()).await;  
 assert!(created\_book2\_result.is\_err());  
 }  
}

The code starts with some import statements and declarations of external dependencies and custom modules.

* The Repository struct is defined with a single field database\_connection, which is an Arc (atomic reference-counted) wrapper around a DatabaseConnection. The Arc allows multiple references to the same connection, facilitating concurrent usage.
* The RepositoryError enum is defined using the thiserror crate to represent errors that can occur during database operations. In this case, it only includes a single variant DatabaseError, which wraps a DbErr (database error) obtained from the sea\_orm library.
* The Repository struct has an implementation block that contains the implementation of its methods.
* The new method is an asynchronous constructor that takes a DatabaseConnection as a parameter and initializes the repository. It performs a database migration using the Migrator from the migration module, handling any errors encountered during migration.
* The create\_book method creates a new book in the database with the given title and ISBN. It constructs a BookActiveModel instance with the provided values and inserts it into the database using the insert method from sea\_orm. Any errors encountered during the database operation are mapped to a RepositoryError.
* Inside the tests module, there are two test functions:
* The test\_create\_book function tests the create\_book method by setting up a PostgreSQL database in a Docker container, establishing a connection to it, creating a repository instance, and calling the create\_book method with some test data. It asserts that the returned book model has the same title and ISBN as the provided input.
* The test\_create\_book\_title\_unique function tests the uniqueness constraint of the title field by creating a book with a specific title and ISBN, and then attempting to create another book with the same title and ISBN. It asserts that the second create operation fails and returns an error.

In summary, the code defines a repository pattern for interacting with a database, specifically for creating books. It provides an asynchronous constructor for initializing the repository and a method for creating books in the database. The tests ensure that the repository functions correctly by setting up a PostgreSQL database in a Docker container and performing various assertions on the repository’s behavior.

**Book Created Kafka Producer**

We can now add the implementation of kafka producer to publish details about the newly created book under src/service/book\_created\_[producer.rs](http://producer.rs/)

use common::events::{  
 constants::Topics,  
 dto::{CreatedBookBuilder, CreatedBookBuilderError},  
};  
use kafka::producer::KafkaProducer;  
use thiserror::Error;  
  
#[derive(Clone)]  
pub struct BookCreatedProducer {  
 producer: KafkaProducer,  
}  
  
#[derive(Error, Debug)]  
pub enum BookCreatedProducerError {  
 #[error("CreatedBookBuilderError error")]  
 CreatedBookBuilderError(#[from] CreatedBookBuilderError),  
}  
  
impl BookCreatedProducer {  
 pub fn new(bootstrap\_servers: String, schema\_registry\_url: String) -> Self {  
 Self {  
 producer: KafkaProducer::new(  
 bootstrap\_servers,  
 schema\_registry\_url,  
 Topics::BookCreated.to\_string(),  
 ),  
 }  
 }  
  
 pub async fn publish\_created\_book(  
 &self,  
 id: i32,  
 title: String,  
 isbn: String,  
 ) -> Result<bool, BookCreatedProducerError> {  
 let created\_book = CreatedBookBuilder::default()  
 .id(id)  
 .title(title)  
 .isbn(isbn)  
 .build()  
 .map\_err(|e| BookCreatedProducerError::CreatedBookBuilderError(e))?;  
 Ok(self.producer.produce(id.to\_string(), created\_book).await)  
 }  
}

1. The code begins with some import statements, including dependencies from the common and kafka modules.
2. The BookCreatedProducer struct is defined with a single field producer, which is an instance of KafkaProducer responsible for sending messages to Kafka.
3. The BookCreatedProducerError enum is defined using the thiserror crate to represent errors that can occur during the process of publishing a created book event. It includes a single variant CreatedBookBuilderError, which wraps an error from the CreatedBookBuilder module.
4. The BookCreatedProducer struct has an implementation block that contains the implementation of its methods.

* The new method is a constructor that takes bootstrap\_servers, schema\_registry\_url, and Topics::BookCreated as parameters. It initializes the producer field with a new instance of KafkaProducer configured with the provided parameters.
* The publish\_created\_book method is an asynchronous method that publishes a book creation event to Kafka. It takes id, title, and isbn as parameters. It uses the CreatedBookBuilder to construct a CreatedBook data transfer object (DTO) with the provided values. If the builder fails to construct the DTO, the error is mapped to a BookCreatedProducerError. Otherwise, the created\_book is passed to the produce method of the producer to publish it to Kafka. The result of the produce operation is returned as a Result<bool, BookCreatedProducerError>.

In summary, the code defines a BookCreatedProducer struct that uses a KafkaProducer to publish book creation events to a Kafka topic. It provides a constructor for initializing the producer and a method for publishing a created book event to Kafka. Any errors encountered during the process are captured and represented as a custom BookCreatedProducerError enum.

**Service**

Update the src/service/[mod.rs](http://mod.rs/) with the below code

pub mod book\_created\_producer;  
use self::book\_created\_producer::{BookCreatedProducer, BookCreatedProducerError};  
use crate::dto::{Book, BookBuilder, BookBuilderError};  
use crate::repository::{Repository, RepositoryError};  
use std::sync::Arc;  
use thiserror::Error;  
use tracing::{info\_span, Instrument};  
  
#[derive(Clone)]  
pub struct Service {  
 repository: Repository,  
 book\_created\_producer: Arc<BookCreatedProducer>,  
}  
  
#[derive(Error, Debug)]  
pub enum ServiceError {  
 #[error("Repository error")]  
 RepositoryError(#[from] RepositoryError),  
  
 #[error("BookBuilder error")]  
 BookBuilderError(#[from] BookBuilderError),  
  
 #[error("BookCreatedProducer error")]  
 BookCreatedProducer(#[from] BookCreatedProducerError),  
}  
  
impl Service {  
 pub fn new(repository: Repository, book\_created\_producer: BookCreatedProducer) -> Self {  
 Self {  
 repository,  
 book\_created\_producer: Arc::new(book\_created\_producer),  
 }  
 }  
  
 pub async fn create\_and\_publish\_book(  
 &self,  
 title: String,  
 isbn: String,  
 ) -> Result<Book, ServiceError> {  
 let span = info\_span!("create\_and\_publish\_book repository create\_book");  
 let created\_book\_model = async move {  
 let created\_book\_model = self  
 .repository  
 .create\_book(title, isbn)  
 .await  
 .map\_err(|e| ServiceError::RepositoryError(e));  
 return created\_book\_model;  
 }  
 .instrument(span)  
 .await?;  
 let producer = self.book\_created\_producer.clone();  
 let created\_book\_id = created\_book\_model.clone().id;  
 let created\_book\_title = created\_book\_model.clone().title;  
 let created\_book\_isbn = created\_book\_model.clone().isbn;  
 let \_ = tokio::task::spawn(async move {  
 let \_ = producer  
 .publish\_created\_book(created\_book\_id, created\_book\_title, created\_book\_isbn)  
 .await;  
 })  
 .instrument(info\_span!("publish\_created\_book"));  
  
 let book = BookBuilder::default()  
 .id(created\_book\_model.id)  
 .title(created\_book\_model.title)  
 .isbn(created\_book\_model.isbn)  
 .build()  
 .map\_err(|e| ServiceError::BookBuilderError(e))?;  
 Ok(book)  
 }  
}

1. The code begins with some import statements, including dependencies from various modules.
2. The Service struct is defined with two fields: repository of type Repository and book\_created\_producer of type Arc<BookCreatedProducer>. The repository field is used to interact with the book repository, while the book\_created\_producer field is used to publish book creation events.
3. The ServiceError enum is defined using the thiserror crate to represent errors that can occur during book creation and publishing. It includes three variants: RepositoryError to wrap errors from the Repository module, BookBuilderError to wrap errors from the BookBuilder module, and BookCreatedProducer to wrap errors from the BookCreatedProducer module.
4. The Service struct has an implementation block that contains the implementation of its methods.

* The new method is a constructor that takes a repository of type Repository and book\_created\_producer of type BookCreatedProducer as parameters. It initializes the respective fields of the Service struct.
* The create\_and\_publish\_book method is an asynchronous method responsible for creating and publishing a book. It takes title and isbn as parameters. It begins by creating an information span for logging purposes. Inside the span, it calls the create\_book method of the repository to create a book model. If an error occurs during the repository operation, it is mapped to a RepositoryError and returned. The book creation and repository operation are instrumented with the span. Next, the book\_created\_producer is cloned to be used later for publishing the created book. The book's ID, title, and ISBN are extracted from the created book model. A new Tokio task is spawned to asynchronously publish the created book using the book\_created\_producer. The task is instrumented with a new information span. Finally, a Book object is constructed using the BookBuilder based on the created book model, and it is returned as a successful result.

In summary, the code defines a Service struct that acts as a service layer for creating and publishing books. It uses a Repository to create books and a BookCreatedProducer to publish book creation events. Errors that can occur during the process are captured and represented as a customServiceError enum.

**HTTP Server**

We can add the HTTP Server implementation which exposes the POST endpoint to create a book which can be called by any client. Add the below code to http\_[server.rs](http://server.rs/)

use crate::service::{Service, ServiceError};  
use axum::{http::StatusCode, response::IntoResponse, routing::post, Extension, Json, Router};  
use axum\_tracing\_opentelemetry::opentelemetry\_tracing\_layer;  
use serde::{Deserialize, Serialize};  
use serde\_json::json;  
use std::net::SocketAddr;  
use tracing::error;  
  
pub async fn start\_http\_server(service: Service) {  
 let books\_router = Router::new().route("/", post(create\_book));  
 let api\_router = Router::new().nest("/books", books\_router);  
 let app = Router::new()  
 .nest("/api", api\_router)  
 .layer(opentelemetry\_tracing\_layer())  
 .layer(Extension(service));  
 let addr = SocketAddr::from(([127, 0, 0, 1], 8080));  
 axum::Server::bind(&addr)  
 .serve(app.into\_make\_service())  
 .await  
 .unwrap()  
}  
  
impl IntoResponse for ServiceError {  
 fn into\_response(self) -> axum::response::Response {  
 error!("Service Error {}", self);  
 let (status, error\_message) = match self {  
 ServiceError::RepositoryError(re) => {  
 (StatusCode::INTERNAL\_SERVER\_ERROR, re.to\_string())  
 }  
 e => (StatusCode::INTERNAL\_SERVER\_ERROR, e.to\_string()),  
 };  
  
 let body = Json(json!({ "error": error\_message }));  
 (status, body).into\_response()  
 }  
}  
  
#[derive(Serialize, Deserialize, Debug)]  
struct CreateBookRequest {  
 title: String,  
 isbn: String,  
}  
  
#[derive(Serialize, Deserialize, Debug)]  
struct CreateBookResponse {  
 id: i32,  
 title: String,  
 isbn: String,  
}  
  
async fn create\_book(  
 Extension(service): Extension<Service>,  
 Json(create\_book\_request): Json<CreateBookRequest>,  
) -> impl IntoResponse {  
 let created\_book\_result = service  
 .create\_and\_publish\_book(create\_book\_request.title, create\_book\_request.isbn)  
 .await;  
 return match created\_book\_result {  
 Ok(created\_book) => (  
 StatusCode::CREATED,  
 Json(json!(CreateBookResponse {  
 id: created\_book.id,  
 title: created\_book.title,  
 isbn: created\_book.isbn,  
 })),  
 ),  
 Err(e) => (  
 StatusCode::INTERNAL\_SERVER\_ERROR,  
 Json(json!({ "error": e.to\_string() })),  
 ),  
 };  
}

1. The code begins with some import statements, including dependencies from various modules.
2. The start\_http\_server function is defined as an entry point for starting the HTTP server. It takes a service of type Service as a parameter. Inside the function, a router is created to handle book-related routes. The /books route is nested inside the /api route. The OpenTelemetry tracing layer is added to the router, and the service is passed as an extension to the router. The server is then bound to a socket address (127.0.0.1:8080), and the router is served using the axum::Server.
3. The IntoResponse trait is implemented for the ServiceError enum. This trait allows converting the ServiceError into an Axum response. Inside the implementation, an error message is logged using tracing::error!. Depending on the type of error, an appropriate HTTP status code and error message are determined. The error message is wrapped in a JSON response body. The resulting status code, body, and headers are returned as an Axum response.
4. Two structs, CreateBookRequest and CreateBookResponse, are defined for the request and response bodies of the create book API.
5. The create\_book function is the handler for the /api/books POST route, responsible for creating a book. It takes an Extension of Service and a Json of CreateBookRequest as parameters. Inside the function, the create\_and\_publish\_book method is called on the service to create and publish the book. The result is matched using a match expression. If the book creation is successful, a CreateBookResponse is constructed with the book's ID, title, and ISBN, and it is returned as a JSON response with the status code 201 CREATED. If an error occurs, an error message is returned as a JSON response with the status code 500 INTERNAL SERVER ERROR.

In summary, the code sets up an HTTP server using the Axum framework to handle book creation requests. It defines routes, request/response structs, and error handling. The start\_http\_server function starts the server, and the create\_book function is the handler for creating books. Errors are logged and converted into appropriate HTTP responses.

**Main**

ASSEMBLE !!

mod dto;  
mod entity;  
mod http\_server;  
mod repository;  
mod service;  
  
use crate::repository::Repository;  
use apache\_avro::AvroSchema;  
use common::events::dto::CreatedBook;  
use database::get\_connection;  
use http\_server::start\_http\_server;  
use kafka::util::register\_schema;  
use opentelemetry::global;  
use service::{book\_created\_producer::BookCreatedProducer, Service};  
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\_api".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 database\_connection =  
 get\_connection("postgres://postgres:postgres@localhost:5433/rust-superapp").await?;  
 let repository = Repository::new(database\_connection.clone())  
 .await  
 .expect("Error creating repository");  
 let schema\_registry\_url = "http://localhost:8081".to\_owned();  
 let book\_created\_producer =  
 BookCreatedProducer::new("localhost:9092".to\_owned(), schema\_registry\_url.clone());  
 let service = Service::new(repository, book\_created\_producer);  
  
 register\_schema(  
 schema\_registry\_url,  
 "book".to\_string(),  
 CreatedBook::get\_schema(),  
 )  
 .await  
 .expect("Error while registering schema");  
  
 start\_http\_server(service).await;  
 global::shutdown\_tracer\_provider();  
 Ok(())  
}

1. The code starts by importing various modules and dependencies.
2. Several modules are declared using the mod keyword, which indicates that they are part of the current crate.
3. 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.
4. The first line in the main function sets the text map propagator for OpenTelemetry to the Zipkin propagator.
5. 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.
6. Next, the tracer is used to create a tracing layer (tracing\_opentelemetry::layer()) that is added to the subscriber.
7. The subscriber is configured with a JSON formatter and an environment filter to determine the log level.
8. The subscriber is then initialized using tracing\_subscriber::registry().init().
9. Database connection is established using the get\_connection function from the database module. The connection string is provided as an argument.
10. A repository instance is created using the Repository::new function, which takes the database connection as an argument.
11. The schema registry URL is assigned to a variable.
12. An instance of BookCreatedProducer is created, passing the Kafka bootstrap servers and the schema registry URL as arguments.
13. An instance of the Service struct is created, passing the repository and book created producer instances.
14. The Avro schema for the CreatedBook event is registered with the schema registry using the register\_schema function.
15. The HTTP server is started by calling the start\_http\_server function, passing the service instance.
16. After the server starts, the global tracer provider is shut down.
17. 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 the application, including the database connection, tracing configuration, repository, Kafka producer, and HTTP server. It initializes the necessary dependencies, registers Avro schema, starts the HTTP server, and shuts down the tracer provider.

We have successfully developed the books\_api, which can create a book and publish a message with schema validation using the schema registry, while ensuring comprehensive tracing. In the next section, we will focus on consuming this data in the books\_analytics module.