Book Management API & UI Using Rust

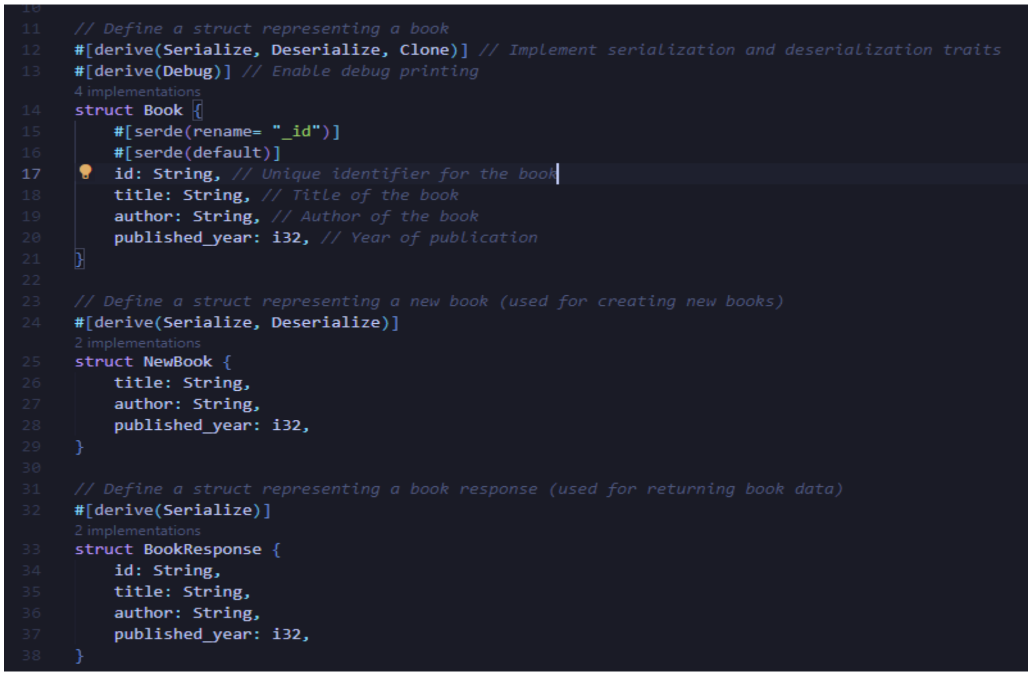
Overview

This project provides a straightforward RESTful API for maintaining a library of books. It offers endpoints for creating, reading, updating, and deleting book records in a MONGODB database. The Rust-based Actix web framework is used to build the API.

Features Of Book Management API

- Create a new book
- Retrieve all books
- Retrieve a single book by ID
- Update a book by ID
- Delete a book by ID
- Serve static HTML and CSS files

Data Structure



```rust
// Define a struct representing a book
#[derive(Serialize, Deserialize, Clone)] // Implement serialization and deserialization traits
#[derive(Debug)] // Enable debug printing
4 implementations
struct Book {
    #[serde(rename= "_id")]
    #[serde(default)]
    id: String, // Unique identifier for the book
    title: String, // Title of the book
    author: String, // Author of the book
    published_year: i32, // Year of publication
}

// Define a struct representing a new book (used for creating new books)
#[derive(Serialize, Deserialize)]
2 implementations
struct NewBook {
    title: String,
    author: String,
    published_year: i32,
}

// Define a struct representing a book response (used for returning book data)
#[derive(Serialize)]
2 implementations
struct BookResponse {
    id: String,
    title: String,
    author: String,
    published_year: i32,
}
```

The Book struct represents a book in the database and has fields for the ID, title, author, and publication year. The NewBook struct is used to create new books, and it has the same fields as Book with the exception of the ID. The BookResponse struct is used to return book data in API responses.

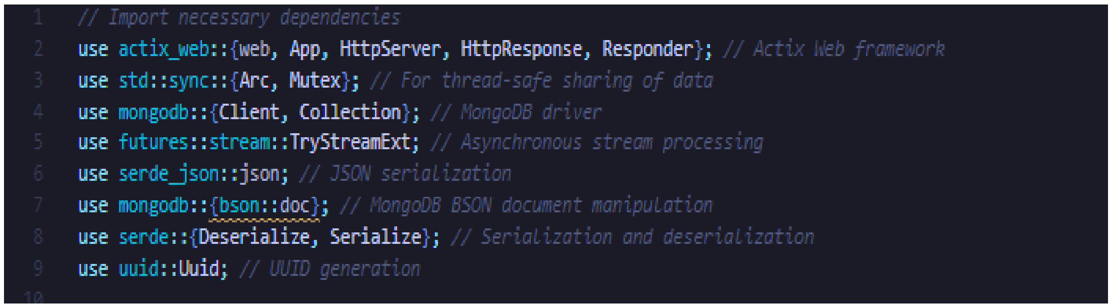
API Handler

The code then defines a few asynchronous methods that serve as handlers for various API endpoints:

create_book: Oversees the creation of a new book. It validates the input data, creates a new UUID for the book's ID, and adds it into the database.

get_all_books retrieves all books from the database and returns them as a JSON array.
get_book_by_id: Retrieves a book's ID from the database and returns it as a JSON object.
update_book: This function updates a book in the database with fresh data from the request body.
Delete_book: Removes a book from the database based on its ID.

Code Explanation

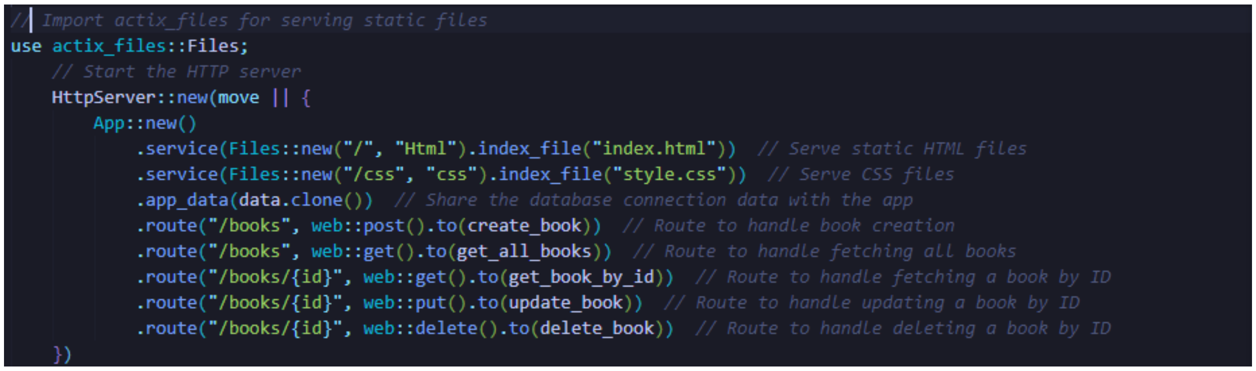


```
// Import necessary dependencies
use actix_web::{web, App, HttpServer, HttpResponse, Responder}; // Actix Web framework
use std::sync::{Arc, Mutex}; // For thread-safe sharing of data
use mongodb::{Client, Collection}; // MongoDB driver
use futures::stream::TryStreamExt; // Asynchronous stream processing
use serde_json::json; // JSON serialization
use mongodb::{bson::doc}; // MongoDB BSON document manipulation
use serde::{Deserialize, Serialize}; // Serialization and deserialization
use uuid::Uuid; // UUID generation
```

These are the most important modules for building web server, handling HTTP request and Response, generating UUID and managing concurrency

Static File Serving

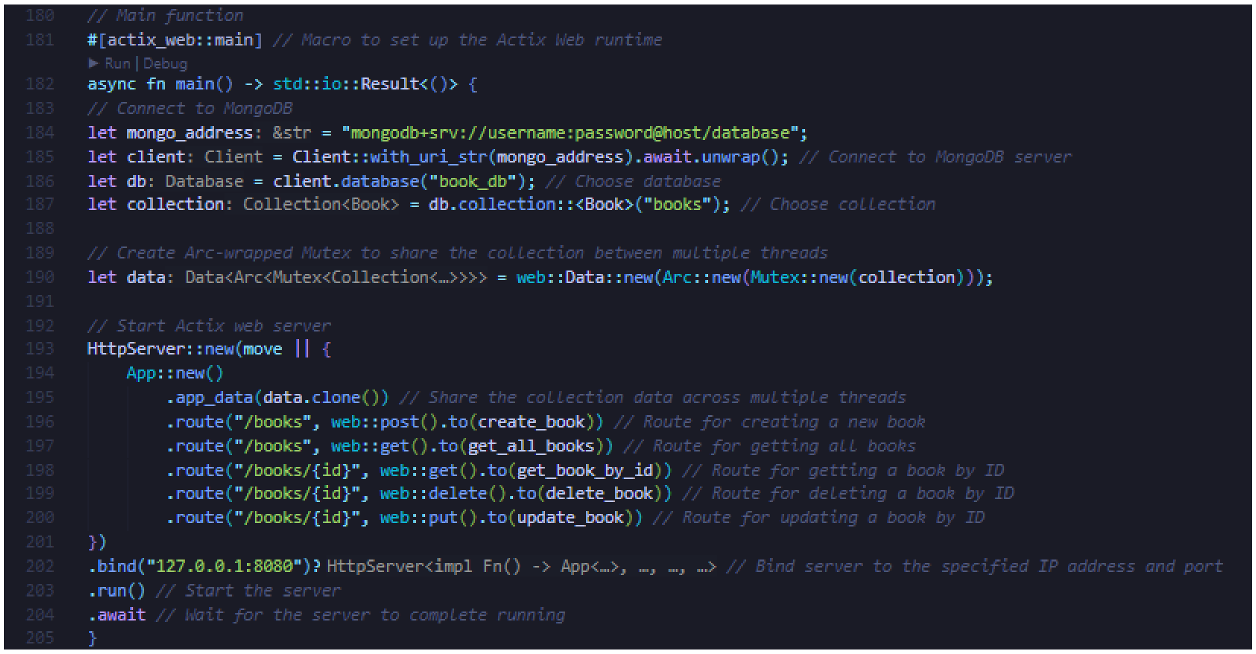
The application supplies static files (HTML and CSS) for the frontend:



```
// Import actix_files for serving static files
use actix_files::Files;
    // Start the HTTP server
    HttpServer::new(move || {
        App::new()
            .service(Files::new("/", "Html").index_file("index.html"))  // Serve static HTML files
            .service(Files::new("/css", "css").index_file("style.css"))  // Serve CSS files
            .app_data(data.clone())  // Share the database connection data with the app
            .route("/books", web::post().to(create_book))  // Route to handle book creation
            .route("/books", web::get().to(get_all_books))  // Route to handle fetching all books
            .route("/books/{id}", web::get().to(get_book_by_id))  // Route to handle fetching a book by ID
            .route("/books/{id}", web::put().to(update_book))  // Route to handle updating a book by ID
            .route("/books/{id}", web::delete().to(delete_book))  // Route to handle deleting a book by ID
    })
```

Main Function

In the main method, we connect to the MongoDB database using the connection string that is supplied. Then we generate a database handle and a collection handle for the "books" collection. Next, we use web Data to construct shared data and send the collection handle to the web server. This enables handlers to access the database collection. Finally, we start the web server with HttpServer new and set up the routes for various API endpoints. The server listens at 127.0.0.1:8080. That is the explanation for the provided code. It configures a Rust web server with Actix-Web and interfaces with a MongoDB database to handle books.



```rust
// Main function
#[actix_web::main] // Macro to set up the Actix Web runtime
async fn main() -> std::io::Result<()> {
// Connect to MongoDB
let mongo_address: &str = "mongodb+srv://username:password@host/database";
let client: Client = Client::with_uri_str(mongo_address).await.unwrap(); // Connect to MongoDB server
let db: Database = client.database("book_db"); // Choose database
let collection: Collection<Book> = db.collection::<Book>("books"); // Choose collection

// Create Arc-wrapped Mutex to share the collection between multiple threads
let data: Data<Arc<Mutex<Collection<…>>>> = web::Data::new(Arc::new(Mutex::new(collection)));

// Start Actix web server
HttpServer::new(move || {
    App::new()
        .app_data(data.clone()) // Share the collection data across multiple threads
        .route("/books", web::post().to(create_book)) // Route for creating a new book
        .route("/books", web::get().to(get_all_books)) // Route for getting all books
        .route("/books/{id}", web::get().to(get_book_by_id)) // Route for getting a book by ID
        .route("/books/{id}", web::delete().to(delete_book)) // Route for deleting a book by ID
        .route("/books/{id}", web::put().to(update_book)) // Route for updating a book by ID
})
.bind("127.0.0.1:8080")? HttpServer<impl Fn() -> App<…>, …, …, …> // Bind server to the specified IP address and port
.run() // Start the server
.await // Wait for the server to complete running
}
```

Problems & Solution

There were a lot of problems I faced in creating task.

- **Serving Static file**:

It was difficult to include the capability of serving static files in addition to the API endpoints. The Actix web server needed to be set up so that HTML and CSS files could be served from particular directories. To serve static files, I utilized the actix_files crate. I was able to provide the required frontend files in the Actix App by configuring the Files services.