Demo of Rust and axum web framework

Demonstration of:

- Rust: programming language that focuses on reliability and stability.
- axum: web framework that focuses on ergonomics and modularity.
- Tower: library of modular and reusable components for building robust clients and servers.
- Hyper: fast and safe HTTP library for the Rust language.
- Tokio: event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications.
- Serde: serialization/deserialization framework.

Thanks to all the above projects and their authors. Donate to them if you can.

Does this demo help your work? Donate here if you can via GitHub sponsors.

Have an idea, suggestion, or feedback? Let us know via GitHub issues.

1. Introduction

axum is a web application framework that focuses on ergonomics and modularity.

Features include:

- Web request routes, async handlers, and parsers/extractors.
- Simple predictable error handling, such as for HTTP 404 not found.
- Use of ecosystems of tower, hyper, and tokio.

This demo shows how to:

- Create a project using Rust and the axum web framework.
- Create axum router routes and their handler functions.
- Create responses with HTTP status code OK and HTML text.
- Create a binary image and respond with a custom header.
- Create functionality for HTTP GET, PUT, PATCH, POST, DELETE.
- Use axum extractors for query parameters and path parameters.
- Create a data store and access it using RESTful routes.

2. Hello, World!

Create a typical new Rust project:

```
cargo new demo-rust-axum
cd demo-rust-axum
```

Edit file `Cargo.toml`.

Use this kind of package and these dependencies:

```
[package]
name = "demo-rust-axum"
version = "0.1.0"
edition = "2021"

[dependencies]
axum = "0.4.8" # Web framework that focuses on ergonomics and modularity.
hyper = { version = "0.14.17", features = ["full"] } # A fast and correct HTTP library.
tokio = { version = "1.17.0", features = ["full"] } # Event-driven, non-blocking I/O platform.
tower = "0.4.12" # Modular reusable components for building robust clients and servers.
serde = { version = "1.0.136", features = ["derive"] } # A serialization/deserialization framework.
serde_json = "1.0.79" # Serde serializion/deserialization of JSON data.
```

Edit file `src/main.rs`.

```
#[tokio::main]
pub async fn main() {
     // Build our application by creating our router.
    let app = axum::Router::new()
        .route("/", axum::routing::get(|| async { "Hello, World!" }));

    // Run our application by using hyper and URL http://localhost:3000.
    // The `Server` is a hyper server, which means you can use any hyper
    // server functions, such as `bind`, `with_graceful_fallback`, etc.
    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(app.into_make_service())
        .await
        .unwrap();
}
```

Try the demo...

Shell:

`cargo run`

Browse http://localhost:3000

You should see "Hello, World!".

In your shell, press CTRL-C to shut down.

3. Graceful shutdown

We want our demo server to be able to do graceful shutdown.

Tokio graceful shutdown generally does these steps:

- Find out when to shut down.
- Tell each part of the program to shut down.
- Wait for each part of the program to shut down.
- [Read tokio documentation about graceful shutdown]https://tokio.rs/tokio/topics/shutdown

Hyper graceful shutdown generally does these steps:

- The server stops accepting new requests.
- The server waits for all in-progress requests to complete.
- Then the server shuts down.
- Read hyper documentation about graceful shutdown

Edit file `main.rs`.

Create a tokio signal handler that listens for a user pressing CTRL+C:

```
/// Tokio signal handler that will wait for a user to press CTRL+C.
/// We use this in our hyper `Server` method `with_graceful_shutdown`.
async fn signal_shutdown() {
    tokio::signal::ctrl_c()
        .await
        .expect("expect tokio signal ctrl-c");
    println!("signal shutdown");
}
```

Modify the `axum::Server` code to add the method `with_graceful_shutdown`:

```
axum::Server::bind(&addr)
    .serve(app.into_make_service())
    .with_graceful_shutdown(shutdown_signal())
```

```
    .await
    .unwrap();
```

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000

You should see "Hello, World!".

In your shell, press CTRL-C.

Your shell should print "^Csignal shutdown" or possibly just "Csignal shutdown".

4. Create a new route and handler function

An axum route can call an function, which is called an axum handler. The handler is async function returns something that can be converted into a response.

Edit file `main.rs`.

The demo will use the axum routing `get` function, quite often, so add code to use it:

```
use axum::routing::get;
```

Add a handler, which is an async function that returns a string:

```
/// axum handler for "GET /" which returns a string, which causes axum to
/// immediately respond with a `200 OK` response, along with the plain text.
pub async fn hello() -> String {
   "Hello, World!".to_string()
}
```

Modify the `Router` code like this:

```
let app = Router::new()
    .route("/", get(hello));
```

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000

You should see "Hello, World!".

5. Create a route that responds with a HTML file

Create file `hello.html`.

Add this:

```
<h1>Hello</h1>
This is our demo.
```

Edit file `main.rs`.

Add route:

```
let app = Router::new()
    ...
    .route("/hello.html", get(hello_html))
```

Add handler:

```rust
/// axum handler that responds with a typical HTML file.
/// This uses the Rust `std::include_str` macro to include a UTF-8 file
/// as `&'static str` in compile time; the path is relative to `main.rs`.
/// Credit <https://github.com/programatik29/axum-tutorial>
async fn hello_html() -> axum::response::Html<&'static str> {
    include_str!("hello.html").into()
}
```

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000/hello.html

You should see the headline "Hello" and text "This is our demo.".

6. Create a router fallback response "not found"

For a request that fails to match anything in the router, you can use the function `fallback`.

Edit file `main.rs`.

Add code for the fallback handler trait:

```rust
use axum::handler::Handler;
```

Modify the `Router` to add the function `fallback` as the first choice:

```rust
let app = Router::new()
    .fallback(fallback.into_service()),
    .route("/", get(hello));
```

Add the `fallback` handler:

```rust
/// axum handler for any request that fails to match the router routes.
/// This implementation returns a HTTP status code 404 Not Found response.
pub async fn fallback(uri: axum::http::Uri) -> impl axum::response::IntoResponse {
    (axum::http::StatusCode::NOT_FOUND, format!("No route for {}", uri))
}
```

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000/whatever

You should see "No route for /whatever".

7. Create a response with HTTP status code OK

Edit file `main.rs`.

Add code to use `StatusCode`:

```rust
use axum::{
    ...
    http::StatusCode,
};
```

Add a route:

```rust
let app = Router::new()
    ...
    .route("/demo-status", get(demo_status));
```

Add a handler:

```rust
/// axum handler for "GET /demo-status" which returns a HTTP status code, such
/// as HTTP status code 200 OK, and an arbitrary user-visible string message.
pub async fn demo_status() -> (axum::http::StatusCode, String) {
    (axum::http::StatusCode::OK, "Everything is OK".to_string())
}
```

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000/demo-status

You should see "Everything is OK".

8. Create a response that echos the URI

Edit file `main.rs`.

Add a route:

```rust
let app = Router::new()
    ...
    .route("/demo-uri", get(demo_uri));
```

Add a handler:

```rust
/// axum handler for "GET /demo-uri" which shows the request's own URI.
/// This shows how to write a handler that receives the URI.
pub async fn demo_uri(uri: axum::http::Uri) -> String {
    format!("The URI is: {:?}", uri)
}
```

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000/demo-uri

You should see "The URI is: /demo-uri!".

9. Create routes and handlers for HTTP verbs

axum routes can use HTTP verbs, including GET, PUT, PATCH, POST, DELETE.

Edit file `main.rs`.

Add axum routes for each HTTP verb:

```rust
let app = Router::new()
    ...
    .route("/foo", get(get_foo).put(put_foo).patch(patch_foo).post(post_foo).delete(delete_foo))
```

Add axum handlers:

```rust
/// axum handler for "GET /foo" which returns a string message.
/// This shows our naming convention for HTTP GET handlers.
pub async fn get_foo() -> String {
   "GET foo".to_string()
}

/// axum handler for "PUT /foo" which returns a string message.
/// This shows our naming convention for HTTP PUT handlers.
pub async fn put_foo() -> String {
   "PUT foo".to_string()
}
```

```rust
/// axum handler for "PATCH /foo" which returns a string message.
/// This shows our naming convention for HTTP PATCH handlers.
pub async fn patch_foo() -> String {
    "PATCH foo".to_string()
}

/// axum handler for "POST /foo" which returns a string message.
/// This shows our naming convention for HTTP POST handlers.
pub async fn post_foo() -> String {
    "POST foo".to_string()
}

/// axum handler for "DELETE /foo" which returns a string message.
/// This shows our naming convention for HTTP DELETE handlers.
pub async fn delete_foo() -> String {
    "DELETE foo".to_string()
}
```

Try the demo...

Shell:

```
cargo run
```

To make a request using an explicit request of GET or POST or DELETE, one way is to use a command line program such as `curl` like this:

Shell:

```
curl --request GET 'http://localhost:3000/foo'
```

Output:

```
GET foo
```

Shell:

```
curl --request PUT 'http://localhost:3000/foo'
```

Output:

```
PUT foo
```

Shell:

```
curl --request PATCH 'http://localhost:3000/foo'
```

Output:

```
PATCH foo
```

Shell:

```
curl --request POST 'http://localhost:3000/foo'
```

Output:

```
POST foo
```

Shell:

```
curl --request DELETE 'http://localhost:3000/foo'
```

Output:

```
DELETE foo
```

The command `curl` uses GET by default i.e. these are equivalent:

```
curl 'http://localhost:3000/foo'
```

```
curl --request GET 'http://localhost:3000/foo'
```

10. Create a response with HTML text

Edit file `main.rs`.

Add code to use `Html`:

```
use axum::{
    ...
    response::Html,
};
```

Add a route:

```
let app = Router::new()
    ...
    .route("/demo.html", get(get_demo_html));
```

Add a handler:

```
/// axum handler for "GET /demo.html" which responds with HTML text.
/// The `Html` type sets an HTTP header content-type of `text/html`.
pub async fn get_demo_html() -> axum::response::Html<&'static str> {
    "<h1>Hello</h1>".into()
}
```

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000/demo.html

You should see HTML with headline text "Hello".

11. Create a response with an image and header

Edit file `Cargo.toml`.

Add dependencies:

```
base64 = "0.13" # Encode and decode base64 as bytes or utf8.
http = "0.2.6" # Types for HTTP requests and responses.
```

Edit file `main.rs`.

Add a route:

```
let app = Router::new()
    ...
    .route("/demo.png", get(get_demo_png))
```

Add a handler:

```
/// axum handler for "GET /demo.png" which responds with a PNG and header.
/// This creates an image, then responds with a new header "image/png".
/// Credit <https://github.com/ttys3/static-server/blob/main/src/main.rs>
async fn get_demo_png() -> impl axum::response::IntoResponse {
    let png = "iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAADUlEQVR42mPk+89QDwADvgGOSHzRgAAAAABJRU5E
    let body = axum::body::Full::from(::base64::decode(png).unwrap());
    let mut response = axum::response::Response::new(body);
    response.headers_mut().insert(
        ::http::header::CONTENT_TYPE,
        ::http::header::HeaderValue::from_static("image/png")
    );
    response
}
```

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000/demo.png

You browser should download a a one-pixel transparent PNG image.

12. Create a route that gets JSON data

axum has capabilties for working with JSON data.

The axum extractor for JSON can also help with a request, by deserializing a request body into some type that implements serde::Deserialize. If the axum extractor is unable to parse the request body, or the request does not contain the Content-Type: application/json header, then the axum extractor will reject the request and return a 400 Bad Request response.

The axum extractor for JSON can help with a response, by formating JSON data then setting the response application content type.

Edit file `main.rs`.

Add code to use Serde JSON:

```
/// Use Serde JSON to serialize/deserialize JSON, such as the request body.
/// axum creates JSON payloads or extracts them by using `axum::extract::Json`.
/// For the implementation, see functions `get_demo_json` and `post_demo_json`.
use serde_json::{json, Value};
```

Add a route:

```
let app = Router::new()
    ...
    .route("/demo-json", get(get_demo_json));
```

Add a handler:

```
/// axum handler for "GET /demo.json" which shows how to return JSON data.
/// The `Json` type sets an HTTP header content-type of `application/json`.
/// The `Json` type works with any type that implements `serde::Serialize`.
pub async fn get_demo_json() -> axum::extract::Json<Value> {
    json!({"a":"b"}).into()
}
```

Try the demo...

Shell:

```
cargo run
```

To request JSON with curl, set a custom HTTP header like this:

```
curl \
--header "Accept: application/json" \
--request GET 'http://localhost:3000/demo-json'
```

Output:

```
{"a":"b"}
```

13. Create a route that extracts its JSON payload

Edit file `main.rs`.

Add code to use `Json`:

```
use axum::{
    ...
```

```
    extract::Json,
};
```

Modify the route `/demo.json` to append the function `put`:

```
let app = Router::new()
    ...
    .route("/demo.json", get(get_demo_json).put(put_demo_json))
```

Add a handler:

```
/// axum handler for "PUT /demo-json" which shows how to use `aumx::extract::Json`.
/// This buffers the request body then deserializes it into a `serde_json::Value`.
/// The axum `Json` type supports any type that implements `serde::Deserialize`.
pub async fn put_demo_json(axum::extract::Json(payload): axum::extract::Json<serde_json::Value>) -> Str
    format!("Put demo JSON payload: {:?}", payload)
}
```

Try the demo...

Shell:

```
cargo run
```

Send the JSON:

```
curl \
--request PUT 'http://localhost:3000/demo-json' \
--header "Content-Type: application/json" \
--data '{"a":"b"}'
```

Output:

```
Put demo JSON payload: Object({"a": String("b")})
```

14. Create a route that extracts query parameters

An axum "extractor" is how you pick apart the incoming request in order to get any parts that your handler needs.

Edit file `main.rs`.

Add code to use HashMap to deserialize query parameters into a key-value map:

```
use std::collections::HashMap;
```

Add a route:

```
let app = Router::new()
    ...
    .route("/items", get(get_items));
```

Add a handler:

```
/// axum handler for "GET /item" which shows how to use `axum::extrac::Query`.
/// This extracts query parameters then deserializes them into a key-value map.
pub async fn get_items(axum::extract::Query(params): axum::extract::Query<HashMap<String, String>>) ->
    format!("Get items with query params: {:?}", params)
}
```

Try the demo...

Shell:

```
cargo run
```

Shell:

```
curl 'http://localhost:3000/items?a=b'
```

Output:

```
Get items with query params: {"a": "b"}
```

15. Create a route that extracts path parameters

Add a route using path parameter syntax, such as ":id", in order to tell axum to extract a path parameter and deserialize it into a variable named `id`.

Edit file `main.rs`.

Add a route:

```
let app = Router::new()
    ...
    .route("/items/:id", get(get_items_id));
```

Add a handler:

```
/// axum handler for "GET /items/:id" which shows how to use `axum::extract::Path`.
/// This extracts a path parameter then deserializes it as needed.
pub async fn get_items_id(axum::extract::Path(id): axum::extract::Path<String>) -> String {
    format!("Get items with path id: {:?}", id)
}
```

Try the demo...

Shell:

```
cargo run
```

Shell:

```
curl 'http://localhost:3000/items/1'
```

Ouput:

```
Get items with id: 1
```

16. Create a book struct

Suppose we want our app to have features related to books.

Create a new file `book.rs`.

Add code to use deserialization:

```
/// Use Deserialize to convert e.g. from request JSON into Book struct.
use serde::Deserialize;
```

Add code to create a book struct that derives the traits we want:

```
/// Demo book structure with some example fields for id, title, author.
/// A production app or database could use an id that is a u32, UUID, etc.
#[derive(Debug, Deserialize, Clone, Eq, Hash, PartialEq)]
pub struct Book {
    pub id: String,
    pub title: String,
    pub author: String,
}
```

Add code to implement `Display`:

```
/// Display the book using the format "{title} by {author}".
/// This is a typical Rust trait and is not axum-specific.
impl std::fmt::Display for Book {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "{} by {}", self.title, self.author)
    }
}
```

Edit file `main.rs`.

Add code to include the `book` module and use the `Book` struct:

```
/// See file book.rs, which defines the `Book` struct.
mod book;
use crate::book::Book;
```

17. Create a data store

For a production app, we could implement the data by using a database.

For this demo, we will implement the data by using a global variable `DATA`.

Edit file `Cargo.toml`.

Add the dependency `once_cell` which is for our global variables:

```
once_cell = "1.10.0" # Single assignment cells and lazy values.
```

Create file `data.rs`.

Add this code:

```
/// Use once_cell for creating a global variable e.g. our DATA data.
use once_cell::sync::Lazy;

/// Use Mutex for thread-safe access to a variable e.g. our DATA data.
use std::sync::Mutex;

/// Create a data store as a global variable with `Lazy` and `Mutex`.
/// This demo implementation uses a `HashMap` for ease and speed.
/// The map key is a primary key for lookup; the map value is a Book.
///
/// To access data, create a thread, spawn it, and acquire the lock:
/// ```
/// async fn example() {
///     thread::spawn(move || {
///         let data = DATA.lock().unwrap();
///         ...
/// }).join().unwrap()
/// ```
static DATA: Lazy<Mutex<HashMap<u32, Book>>> = Lazy::new(|| Mutex::new(
    HashMap::from([
        (1, Book { id: 1, title: "Antigone".into(), author: "Sophocles".into()}),
        (2, Book { id: 2, title: "Beloved".into(), author: "Toni Morrison".into()}),
        (3, Book { id: 3, title: "Candide".into(), author: "Voltaire".into()}),
    ])
));
```

Edit file `main.rs`.

Add code to include the `data` module and use the `DATA` global variable:

```
/// See file data.rs, which defines the DATA global variable.
mod data;
use crate::data::DATA;

/// Use Thread for spawning a thread e.g. to acquire our DATA mutex lock.
use std::thread;
```

18. Create a route to get all books

Edit file `main.rs`.

Add a route:

```
let app = Router::new()
    ...
    .route("/books", get(get_books));
```

Add a handler:

```
/// axum handler for "GET /books" which returns a resource index HTML page.
/// This demo uses our DATA variable; a production app could use a database.
/// This function needs to clone the DATA in order to sort them by title.
pub async fn get_books() -> axum::response::Html<String> {
    thread::spawn(move || {
        let data = DATA.lock().unwrap();
        let mut books = data.values().collect::<Vec<_>>().clone();
        books.sort_by(|a, b| a.title.cmp(&b.title));
        books.iter().map(|&book|
            format!("<p>{}</p>\n", &book)
        ).collect::<String>()
    }).join().unwrap().into()
}
```

Try the demo...

Shell:

```
cargo run
```

Shell:

```
curl 'http://localhost:3000/books'
```

Output:

```
<p>Antigone by Sophocles</p>
<p>Beloved by Toni Morrison</p>
<p>Candide by Voltaire</p>
```

19. Create a route to put a book

Edit file `main.rs`.

Modify the route `/books` to append the function `put`:

```
let app = Router::new()
    ...
    .route("/books", get(get_books).put(put_books));
```

Add a handler:

```
/// axum handler for "PUT /books" which creates a new book resource.
/// This demo shows how axum can extract a JSON payload into a Book struct.
pub async fn put_books(axum::extract::Json(book): axum::extract::Json<Book>) -> axum::response::Html<St
    DATA.lock().unwrap().insert(book.id, book.clone());
    format!("Put book: {}", &book).into()
}
```

Try the demo...

Shell:

```
cargo run
```

Shell:

```
curl \
--request PUT 'http://localhost:3000/books' \
--header "Content-Type: application/json" \
--data '{"id":"4","title":"Decameron","author":"Giovanni Boccaccio"}'
```

Output:

```
Put book: Decameron by Giovanni Boccaccio
```

Shell:

```
curl 'http://localhost:3000/books'
```

Output:

```
<p>Antigone by Sophocles</p>
<p>Beloved by Toni Morrison</p>
<p>Candide by Voltaire</p>
<p>Decameron by Giovanni Boccaccio</p>
```

20. Create a route to get one book id

Edit file `main.rs`.

Add a route:

```
let app = Router::new()
    ...
    .route("/books/:id", get(get_books_id));
```

Add a handler:

```
/// axum handler for "GET /books/:id" which responds with one resource HTML page.
/// This demo app uses our DATA variable, and iterates on it to find the id.
pub async fn get_books_id(axum::extract::Path(id): axum::extract::Path<u32>) -> axum::response::Html<St
    match DATA.lock().unwrap().get(&id) {
        Some(book) => format!("<p>{}</p>\n", &book),
        None => format!("<p>Book id {} not found</p>", id),
    }.into()
}
```

Try the demo...

Shell:

```
cargo run
```

Shell:

```
curl 'http://localhost:3000/books/1'
```

Output:

```
<p>Antigone by Sophocles</p>
```

Shell:

```
curl 'http://localhost:3000/books/0'
```

Output:

```
<p>Book id 0 not found</p>
```

21. Create a route to delete one book id

Edit file `main.rs`.

Modify the route `/books/:id` to append the function `delete`:

```
let app = Router::new()
    ...
    .route("/books/:id", get(get_books_id).delete(delete_books_id));
```

Add a handler:

```
/// axum handler for "DELETE /books/:id" which destroys an existing resource.
/// This code shows how to extract an id, then mutate the DATA variable.
pub async fn delete_books_id(axum::extract::Path(id): axum::extract::Path<u32>) -> axum::response::Html
    thread::spawn(move || {
```

```
        let mut data = DATA.lock().unwrap();
        if data.contains_key(&id) {
            data.remove(&id);
            format!("Delete book id: {}", &id)
        } else {
            format!("Book id not found: {}", &id)
        }
    }).join().unwrap().into()
}
```

Try the demo...

Shell:

```
cargo run
```

Shell:

```
curl --request DELETE 'http://localhost:3000/books/1'
```

Output:

```
<p>Delete book id: 1</p>
```

Shell:

```
curl 'http://localhost:3000/books'
```

Output:

```
<p>Beloved by Toni Morrison</p>
<p>Candide by Voltaire</p>
```

22. Create a route to get one book as an editable form

Edit file `main.rs`.

Add a route:

```
let app = Router::new()
    ...
    .route("/books/:id/form", get(get_books_id_form));
```

Add a handler:

```
/// axum handler for "GET /books/:id/form" which responds with an HTML form.
/// This demo shows how to write a typical HTML form with input fields.
pub async fn get_books_id_form(axum::extract::Path(id): axum::extract::Path<u32>) -> axum::response::Ht
    match DATA.lock().unwrap().get(&id) {
        Some(book) => format!(
            concat!(
                "<form method=\"post\" action=\"/books/{}/form\">\n",
                "<input type=\"hidden\" name=\"id\" value=\"{}\">\n",
                "<p><input type=\"text\" name=\"title\" value=\"{}\"></p>\n",
                "<p><input type=\"text\" name=\"author\" value=\"{}\"></p>\n",
                "<input type=\"submit\" value=\"Save\">\n",
                "</form>\n"
            ),
            &book.id,
            &book.id,
            &book.title,
            &book.author
        ),
        None => format!("<p>Book id {} not found</p>", id),
    }.into()
}
```

Try the demo...

Shell:

```
cargo run
```

Shell:

```
curl 'http://localhost:3000/books/1/form'
```

Output:

```
<form method="post" action="/books/1/form">
<p><input name="title" value="Antigone"></p>
<p><input name="author" value="Sophocles"></p>
<input type="submit" value="Save">
</form>
```

23. Create a route to submit the form to update a book

Edit file `main.rs`.

Modify the route `/books/:id/form` to append the function `post`:

```
let app = Router::new()
    ...
    .route("/books/:id/form", get(get_books_id_form).post(post_books_id_form));
```

Add a handler:

```
/// axum handler for "POST /books/:id/form" which submits an HTML form.
/// This demo shows how to do a form submission then update a resource.
pub async fn post_books_id_form(form: axum::extract::Form<Book>) -> axum::response::Html<String> {
    let new_book: Book = form.0;
    thread::spawn(move || {
        let mut data = DATA.lock().unwrap();
        if data.contains_key(&new_book.id) {
            data.insert(new_book.id, new_book.clone());
            format!("<p>{}</p>\n", &new_book)
        } else {
            format!("Book id not found: {}", &new_book.id)
        }
    }).join().unwrap().into()
}
```

Try the demo...

Shell:

```
cargo run
```

Shell:

```
curl \
--request POST 'http://localhost:3000/books/1' \
--header "Content-Type: application/json" \
--data '{"id":"1","title":"Antigone and Lysistra","author":"Sophocles of Athens"}'
```

Output:

```
Post book: Antigone and Lysistra by Sophocles of Athens
```

Shell:

```
curl 'http://localhost:3000/books'
```

Output:

```
<p>Antigone and Lysistra by Sophocles of Athens</p>
<p>Beloved by Toni Morrison</p>
<p>Candide by Voltaire</p>
```

24. Bonus: Add a Tower tracing subscriber

Edit file `Cargo.toml`.

Add dependencies:

```
tracing = "0.1.32" #  Application-level tracing for Rust.
tracing-subscriber = { version = "0.3.9", features = ["env-filter"] } # Utilities for tracing.
```

Edit file `main.rs`.

Add code to use tracing:

```
/// Use tracing crates for application-level tracing output.
use tracing_subscriber::{
    layer::SubscriberExt,
    util::SubscriberInitExt,
};
```

Add a tracing subscriber:

```
pub async fn main() {
    // Start tracing.
    tracing_subscriber::registry()
        .with(tracing_subscriber::fmt::layer())
        .init();
    ...
```

Try the demo...

Shell:

```
cargo run
```

You should see console output that shows tracing initialization such as:

```
2022-03-08T00:13:54.483877Z
    TRACE mio::poll:
    registering event source with poller:
    token=Token(1),
    interests=READABLE | WRITABLE
```

25. Bonus: Refactor to use a host, port, and socket address

To bind the server, our demo code uses a socket address string.

Edit file `main.rs`.

The demo code is:

```
axum::Server::bind(&"0.0.0.0:3000".parse().unwrap()) ...
```

If you prefer create a socket address step by step, then you can.

Modify the demo code to do:

```
use std::net::SocketAddr;

pub async fn main() {
    ...
    let host = [127, 0, 0, 1];
    let port = 3000;
    let addr = SocketAddr::from((host, port));
    axum::Server::bind(&addr) ...
```

26. Conclusion: What you learned

You learned how to:

- Create a project using Rust and the axum web framework.

- Create axum router routes and their handler functions.
- Create responses with HTTP status code OK and HTML text.
- Create a binary image and respond with a custom header.
- Create functionality for HTTP GET, PUT, POST, DELETE.
- Use axum extractors for query parameters and path parameters.
- Create a data store and access it using RESTful routes.

27. Epilog: What next

To learn more about Rust and axum:

- The Rust book is an excellent thorough starting point.
- The axum crate has dozens of examples you can try.
- The Tokio website

We welcome constructive feedback via GitHub issues:

- Any ideas for making this demo better?
- Any requests for new demo sections or example topics?
- Any bugs or issues in the demo code or documentation?