Contents

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 for building robust clients and servers.
- hyper: fast and safe HTTP library for the Rust language.
- tokio: platform for writing asynchronous I/O backed applications.
- Serde: serialization/deserialization framework.

Thanks

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.

Feedback

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

Have a code improvement or bug fix? We welcome GitHub pull requests.

License

Contact

Have feedback? Have thoughts about this? Want to contribute?

Contact the maintainer at joel@joelparkerhenderson.com

What is this?

This demo is a tutorial that teaches how to build features from the groung up with axum and its ecosystem of tower middleware, hyper HTTP library, tokio asynchronous platform, and Serde data conversions.

What will you learn?

- Create a project using Rust and the axum web framework.
- Leverage capabilties of a hyper server and tower middleware.
- 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.
- Handle HTTP verbs including GET, PUT, PATCH, POST, DELETE.
- Use axum extractors for query parameters and path parameters.
- Manage a data store and access it using RESTful routes.

What is required?

Some knowledge of Rust programming is required, such as:

- How to create a Rust project, build it, and run it.
- How to write functions and their parameters
- How to use shell command line tools such as curl.

Some knowledge about web frameworks is required, such as:

- The general concepts of HTTP requests and responses.
- The general concepts of of RESTful routes and resources.
- The general concepts of formats for HTML, JSON, and text.

What is helpful?

Some knowledge of web frameworks is hepful, such as:

- Rust web frameworks, such as Actix, Rocket, Warp, etc.
- Other languages' web frameworks, such as Rails, Phoenix, Express, etc.
- Other web-related frameworks, such as React, Vue, Svelte, etc.

Some knowledge of this stack can be helpful, such as:

- middleware programming e.g. withv tower
- asynchronous application programming e.g. with tokio
- HTTP services programming e.g. with hyper

What is axum?

High level features:

- Route requests to handlers with a macro free API.
- Declaratively parse requests using extractors.
- Simple and predictable error handling model.
- Generate responses with minimal boilerplate.
- Take full advantage of the tower and its ecosystem.

How is axum special?

The tower ecosystem is what sets axum apart from other frameworks:

- axum doesn't have its own middleware system but instead uses tower::Service.
- axum gets timeouts, tracing, compression, authorization, and more, for free.
- axum can share middleware with applications written using hyper or tonic.

Why learn axum now?

- axum is combines the speed and security of Rust with the power of battle-tested libraries for middleware, asynchronous programming, and HTTP.
- axum is primed to reach developers who are currently using other Rust web frameworks, such as Actix, Rocket, Warp, and others.
- axum is likely to appeal to programmers are seeking afaster web framework and who want closer-to-the-metal capabilties.

Hello, World!

```
#[tokio::main]
async fn main() {
    // Build our application with a single route.
    let app = axum::Router::new().route("/",
        axum::handler::get(|| async { "Hello, World!" }));

    // Run our application as a hyper server on http://localhost:3000.
    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(app.into_make_service())
        .await
        .unwrap();
}
```

What is tower?

Tower is a library of modular and reusable components for building robust networking clients and servers.

Tower aims to make it as easy as possible to build robust networking clients and servers. It is protocol agnostic, but is designed around a request / response pattern. If your protocol is entirely stream based, Tower may not be a good fit.

Service

At Tower's core is the Service trait. A Service is an asynchronous function that takes a request and produces a response.

```
pub trait Service<Request> {
    type Response;
    type Error;
    type Future: Future<Output = Result<Self::Response, Self::Error>>;

    fn poll_ready(
        &mut self,
        cx: &mut Context<'_>,
    ) -> Poll<Result<(), Self::Error>>;

    fn call(&mut self, req: Request) -> Self::Future;
}
```

Call

The most common way to call a service is:

```
use tower::{
    Service,
    ServiceExt,
};

let response = service
    // wait for the service to have capacity
    .ready().await?
    // send the request
    .call(request).await?;
```

What is hyper?

hyper is a fast HTTP implementation written in and for Rust.

- A Client for talking to web services.
- A Server for building those web services.
- Blazing fast* thanks to Rust.
- High concurrency with non-blocking sockets.
- HTTP/1 and HTTP/2 support.

Hyper is low-level

hyper is a relatively low-level library, meant to be a building block for libraries and applications.

If you are looking for a convenient HTTP client, then you may wish to consider reqwest.

If you are looking for a convenient HTTP server, then you may wish to consider warp.

Both are built on top of hyper.

Hello, World!

```
use std::convert::Infallible;

async fn handle(
    _: hyper::Request<Body>
) -> Result<hyper::Response<hyper::Body>, Infallible> {
    Ok(hyper::Response::new("Hello, World!".into()))
}

#[tokio::main]
async fn main() {
    let addr = SocketAddr::from(([127, 0, 0, 1], 3000));

    let make_svc = hyper::service::make_service_fn(|_conn| async {
        Ok::<_, Infallible>(hyper::service::service_fn(handle))
    });

    let server = hyper::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(make_svc);

    if let Err(e) = server.await {
        eprintln!("server error: {}", e);
    }
}
```

What is tokio?

tokio is an asynchronous runtime for the Rust programming language.

- Building blocks for writing network applications.
- Flexibility to target a wide range of systems.
- Memory-safe, thread-safe, and misuse-resistant.

The tokio stack includes:

- Runtime: Including I/O, timer, filesystem, synchronization, and scheduling.
- Hyper: An HTTP client and server library supporting HTTP protocols 1 and 2.
- Tonic: A boilerplate-free gRPC client and server library for network APIS.
- Tower: Modular components for building reliable clients and servers.
- Mio: Minimal portable API on top of the operating-system's evented I/O API.
- Tracing: Unified, structured, event-based, data collection and logging.
- Bytes: A rich set of utilities for manipulating byte arrays.

Demo tokio server

```rust
#[tokio::main]
async fn main() {
    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
        .await
        .unwrap();
    loop {
        let (socket, _address) = listener.accept().await.unwrap();
        tokio::spawn(async move {
            process(socket).await;
        });
    }
}

async fn process(socket: tokio::net::TcpStream) {
    println!("process socket");
}
```

Demo tokio client

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let mut client = client::connect("127.0.0.1:3000").await?;
    println!("connected);
    Ok(())
}
```

What is Serde?

Serde is a framework for serializing and deserializing Rust data structures efficiently and generically.

The Serde ecosystem consists of data structures that know how to serialize and deserialize themselves along with data formats that know how to serialize and deserialize other things.

Serde provides the layer by which these two groups interact with each other, allowing any supported data structure to be serialized and deserialized using any supported data format.

Design

Serde is built on Rust's powerful trait system.

- Serde provides the `Serialize` trait and `Deserialize` trait for data structures.
- Serde provides `derive` attributes, to generate implementations at compile time.
- Serde has no runtime overhead such as reflection or runtime type information.
- In many situations the interaction between data structure and data format can be completely optimized away by the Rust compiler.

Demo of Serde

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point).unwrap();

    // Print {"x":1,"y":2}
    println!("{}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Print Point { x: 1, y: 2 }
    println!("{:?}", deserialized);
}
```

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]
# Web framework that focuses on ergonomics and modularity.
axum = "0.4.8"

# Modular reusable components for building robust clients and servers.
tower = "0.4.12"

# A fast and correct HTTP library.
hyper = { version = "0.14.17", features = ["full"] }

# Event-driven, non-blocking I/O platform.
tokio = { version = "1.17.0", features = ["full"] }

# A serialization/deserialization framework.
serde = { version = "1.0.136", features = ["derive"] }

# Serde serializion/deserialization of JSON data.
serde_json = "1.0.79"
```

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 as a hyper server on http://localhost:3000.
    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.

Create a handler function

An axum route can call an axum handler, which is an async function that returns anything that axum can convert into a response.

Edit file `main.rs` .

Our demos will use the axum routing `get` function, very 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 and causes axum to
/// immediately respond with status code `200 OK` and with the string.
pub async fn hello() -> String {
   "Hello, World!".into()
}
```

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!”.

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

Create a router fallback

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:

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

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

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

Add the `fallback` handler:

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

Try the demo...

Shell:

```
cargo run
```

Browse http://localhost:3000/whatever

You should see ”No route for /whatever”.

Graceful shutdown

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

- Read tokio documentation about graceful shutdown
- Read hyper documentation about 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.

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.

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”.

The whole code

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

#[tokio::main]
pub async fn main() {
     // Build our application by creating our router.
    let app = axum::Router::new()
        .fallback(
            fallback.into_service()
        ),
        .route("/",
            get(hello)
        );

    // Run our application as a hyper server on http://localhost:3000.
    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(app.into_make_service())
        .with_graceful_shutdown(shutdown_signal())
        .await
        .unwrap();
}

/// 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");
}

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

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

Create axum routes and axum handlers

This section shows how to:

- Respond with HTML text
- Respond with a HTML file
- Respond with HTTP status code OK
- Respond with the request URI
- Respond with a custom header and image
- Respond to mutiple HTTP verbs

Respond 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”.

Respond 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:

```
/// axum handler that responds with typical HTML coming from a file.
/// This uses the Rust macro `std::include_str` to include a UTF-8 file
/// path, relative to `main.rs`, as a `&'static str` at compile time.
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.”.

Respond with HTTP status code OK

Edit file `main.rs` .

Add code to use `StatusCode` :

```
use axum::{
    …
    http::StatusCode,
};
```

Add a route:

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

Add a handler:

```
/// axum handler for "GET /demo-status" which returns a HTTP status
/// code, such as OK (200), and a custom 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”.

Respond with the request URI

Edit file `main.rs` .

Add a route:

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

Add a handler:

```
/// 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!”.

Respond with a custom header and image

Edit file `Cargo.toml`.

Add dependencies:

```
# Encode and decode base64 as bytes or utf8.
base64 = "0.13"

# Types for HTTP requests and responses.
http = "0.2.6"
```

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 an image PNG.
/// This sets a header "image/png" then sends the decoded image data.
async fn get_demo_png() -> impl axum::response::IntoResponse {
    let png = concat!(
        "iVBORw0KGgoAAAANSUhEUgAAAAEAAAAB",
        "CAYAAAAfFcSJAAAADUlEQVR42mPk+89Q",
        "DwADvgGOSHzRgAAAAABJRU5ErkJggg=="
    );
    (
        axum::response::Headers([
            (axum::http::header::CONTENT_TYPE, "image/png"),
        ]),
        base64::decode(png).unwrap(),
    )
}
```

Try the demo…

Shell:

```
cargo run
```

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

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

Respond to multiple 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:

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

Add axum handlers:

```
/// 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()
}

/// 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'
```

Extrators

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

This section shows how to:

- Extract path parameters
- Extract query parameters
- Extract a JSON payload
- Respond with a JSON payload

Extract 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 uses `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
```

Extract query parameters

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 /items" which uses `axum::extract::Query`.
/// This extracts query parameters and creates a key-value pair map.
pub async fn get_items(
    axum::extract::Query(params):
        axum::extract::Query<HashMap<String, 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"}
```

Extract a JSON payload

The axum extractor for JSON deserializes a request body into any type that implements `serde::Deserialize` . If the extractor is unable to parse the request body, or if the request is missing the header `Content-Type: application/json` , then the extractor returns HTTP `BAD_REQUEST` (404).

Edit file `main.rs` .

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 uses `aumx::extract::Json`.
/// This buffers the request body then deserializes it using serde.
/// The `Json` type supports types that implement `serde::Deserialize`.
pub async fn put_demo_json(
    axum::extract::Json(data): axum::extract::Json<serde_json::Value>
) -> String{
    format!("Put demo JSON data: {:?}", data)
}
```

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 data: Object({"a": String("b")})
```

Respond with a JSON payload

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 in a request.
/// axum creates JSON or extracts it by using `axum::extract::Json`.
/// For this demo, 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 "PUT /demo-json" which uses `aumx::extract::Json`.
/// This buffers the request body then deserializes it bu using serde.
/// The `Json` type supports types that implement `serde::Deserialize`.
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"}
```

RESTful routes and resources

This section demonstrates how to:

- Create a book struct
- Create the data store
- Use the data store
- Get all books
- Get one book
- Put one book
- Get one book as a web form
- Post one book as a web form
- Delete one book

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 could prefer an id to be type 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;
```

Create the 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:

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

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.
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()
        }),
    ])
));
```

Use the data store

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;

/// To access data, create a thread, spawn it, then get the lock.
/// When you're done, then join the thread with its parent thread.
async fn print_data() {
    thread::spawn(move || {
        let data = DATA.lock().unwrap();
        println!("data: {:?}" ,data);
    }).join().unwrap()
}
```

If you want to see all the data now, then add function to `main` :

```
async fn main() {
    print_data().await;
    …
```

Try the demo...

Shell:

```
cargo run
```

Output:

```
data: {
    1: Book { id: 1, title: "Antigone", author: "Sophocles" },
    2: Book { id: 2, title: "Beloved", author: "Toni Morrison" },
    3: Book { id: 3, title: "Candide", author: "Voltaire" }
}
```

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 responds with a resource page.
/// This demo uses our DATA; a production app could use a database.
/// This demo must clone the DATA in order to sort items 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>
```

Get one book

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<String> {
    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>
```

Put one 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 JSON data into a Book struct.
pub async fn put_books(
    axum::extract::Json(book): axum::extract::Json<Book>
) -> axum::response::Html<String> {
    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>
```

Get one book as a web 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 a 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::Html<String> {
    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 name=\"title\" value=\"{}\"></p>\n",
                "<p><input 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>
```

Post one book as a web form

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":"Another Title","author":"Someone Else"}'
```

Output:

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

Shell:

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

Output:

```
<p>Another Title by Someone Else</p>
<p>Beloved by Toni Morrison</p>
<p>Candide by Voltaire</p>
```

Delete one book

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 a resource.
/// This demo extracts an id, then mutates the book in the DATA store.
pub async fn delete_books_id(
    axum::extract::Path(id): axum::extract::Path<u32>
) -> axum::response::Html<String> {
    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>
```

Extras

This section shows how to:

- Add a Tower tracing subscriber
- Use a host, port, and socket address

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
```

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
```

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) …
```

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.

What's next

To learn more about Rust, axum, tower, hyper, tokio, and Serde:

- The Rust website
- The Rust book are excellent and thorough.
- The book Asynchronous Programming in Rust
- The axum repo and axux crate provide dozens of runnable examples.
- The tower website and tower crate
- The hyper website and hyper crate.
- The tokio website and tokio crate.
- The Serde crate

Feedback

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?

Contact

Joel Parker Henderson

joel@joelparkerhenderson

https://linkedin.com/in/joelparkerhenderson

https://github.com/joelparkerhenderson

axum examples

The axum source code repository includes many project examples, and these examples are fully runnable.

- async-graphql
- chat
- cors
- customize-extractor-error
- customize-path-rejection
- error-handling-and-dependency-injection
- form
- global-404-handler
- graceful-shutdown
- hello-world
- http-proxy
- jwt
- key-value-store
- low-level-rustls
- multipart-form
- oauth
- print-request-response
- prometheus-metrics
- query-params-with-empty-strings
- readme
- reverse-proxy
- routes-and-handlers-close-together
- sessions
- sqlx-postgres
- sse
- static-file-server
- templates
- testing
- tls-rustls
- todos
- tokio-postgres
- tracing-aka-logging
- unix-domain-socket
- validator
- versioning
- websockets