Demo Rust Axum

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.

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

This demo uses the license Creative Commons Share-and-Share-Alike.

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.

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 ground 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 capabilities 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.
- Create a tracing subscriber and emit tracing events.

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 helpful, 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. with tower
- asynchronous application programming e.g. with tokio
- HTTP services programming e.g. with hyper

What is axum?

axum is a web framework with 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 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 a faster web framework and who want closer-to-the-metal capabilities.

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.
    let listener = tokio::net::TcpListener::bind("0.0.0.0:3000").await.unw
    axum::serve(listener, app).await.unwrap();
}
```

Link to examples/axum-hello-world

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(&addr).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

```
#[tokio::main]
async fn main() {
    let listener = tokio::net::TcpListener::bind("localhost: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

```
#[tokio::main]
async fn main() -> Result<()> {
    let mut client = client::connect("localhost: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

```
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

examples/axum-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-examples-axum-hello-world"
version = "1.0.0"
edition = "2024"

[dependencies]
axum = { version = "~0.8.4" } # Web framework that focuses on ergonomics a
tokio = { version = "~1.45.1", features = ["full"] } # Event-driven, non-b
```

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.
    let listener = tokio::net::TcpListener::bind("0.0.0.0:3000").await.unw
    axum::serve(listener, app)
        .await
        .unwrap();
}
```

Try the demo

Shell:

```
cargo run
```

Browse http://localhost:3000 or run:

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

Output:

```
Hello, World!
```

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

Create a fallback router

examples/axum-fallback-router

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

Edit file `main.rs`.

Add a `fallback` handler:

```
/// axum handler for any request that fails to match the router routes.
/// This implementation responds with HTTP status code NOT FOUND (404).
pub async fn fallback(
    uri: axum::http::Uri
) -> impl axum::response::IntoResponse {
    (axum::http::StatusCode::NOT_FOUND, uri.to_string())
}
```

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

```
let app = Router::new()
    .fallback(
        fallback
    )…
```

Try the demo

Shell:

```
cargo run
```

Browse http://localhost:3000/foo or run curl:

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

Output:

```
/foo
```

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

Graceful shutdown

examples/axum-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
- Read axum example 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.

Add a shutdown signal

Add a shutdown signal that handles a user pressing Ctrl+C and/or a Unix terminate signal.

Edit the file `main.rs`.

Add this section:

```
/// Shutdown signal to run axum with graceful shutdown when
/// a user presses Ctrl+C or Unix sends a terminate signal.
pub async fn shutdown_signal() {
    let ctrl_c = async {
        tokio::signal::ctrl_c()
            .await
            .expect("failed to install Ctrl+C handler");
    };

    #[cfg(unix)]
    let terminate = async {
        tokio::signal::unix::signal(tokio::signal::unix::SignalKind::term
            .expect("failed to install signal handler")
            .recv()
            .await;
    };

    #[cfg(not(unix))]
    let terminate = std::future::pending::<()>();

    tokio::select! {
        _ = ctrl_c => {},
        _ = terminate => {},
    }
}
```

Edit the file `main.rs` line `axum::serve` to add the method `with_graceful_shutdown`:

```
axum::serve(listener, app)
    .with_graceful_shutdown(shutdown_signal())
    .await
    .unwrap();
```

Try the demo

Shell:

```
cargo run
```

On your command line, press Ctrl+C.

You should see the server shut down.

Hello handler function

examples/axum-hello-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 often use the axum routing `get` function, so add code to use it:

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

Edit file `main.rs` and the router code to this:

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

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()
}
```

Try the demo

Shell:

```
cargo run
```

Browse http://localhost:3000 or run:

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

Output:

```
Hello, World!
```

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

Epoch handler function

examples/epoch-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` router code to add this route:

```
let app = axum::Router::new()…

    .route("/epoch",
        get(epoch)
    )…
```

Add a handler that returns a Result of Ok(String) or Err(axum::http::StatusCode):

```
/// axum handler for "GET /epoch" which shows the current epoch time.
/// This shows how to write a handler that uses time and can error.
pub async fn epoch() -> Result<String, axum::http::StatusCode> {
    match std::time::SystemTime::now().duration_since(std::time::SystemTi
        Ok(duration) => Ok(format!("{}", duration.as_secs())),
        Err(_) => Err(axum::http::StatusCode::INTERNAL_SERVER_ERROR)
    }
}
```

Try the demo

Shell:

```
cargo run
```

Browse http://localhost:3000/epoch or run:

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

You should see the epoch represented as seconds since January 1 1970 such as:

```
1750938523
```

Uptime handler function

examples/uptime-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`.

Add this anywhere before the function `main`:

```
/// Create the constant INSTANT so the program can track its own uptime.
pub static INSTANT: std::sync::LazyLock<std::time::Instant> = std::sync::L
```

Edit file `main.rs` router code to add this route:

```
let app = axum::Router::new()…

    .route("/uptime",
        get(uptime)
    )…
```

Add a handler that returns the uptime as seconds, such as sixty seconds meaning one minute:

```
/// axum handler for "GET /uptime" which shows the program's uptime durat
/// This shows how to write a handler that uses a global static lazy value
pub async fn uptime() -> String {
    format!("{}", INSTANT.elapsed().as_secs())
}
```

Try the demo

Shell:

```
cargo run
```

Browse http://localhost:3000/uptime or run:

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

You should see a web page that displays the uptime in seconds, such as sixty seconds meaning one minute:

```
60
```

Count handler function

examples/count-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`.

Add this anywhere before the function `main`:

```
/// Create the atomic variable COUNT so the program can track its own cou
pub static COUNT: std::sync::atomic::AtomicUsize = std::sync::atomic::Ato
```

Edit file `main.rs` router code to add this route:

```
let app = axum::Router::new()…

    .route("/count",
        get(count)
    )…
```

Add a handler that does an atomic increment of the count:

```
/// axum handler for "GET /count" which shows the program's count duratio
/// This shows how to write a handler that uses a global static lazy valu
pub async fn count() -> String {
    COUNT.fetch_add(1, std::sync::atomic::Ordering::SeqCst);
    format!("{}", COUNT.load(std::sync::atomic::Ordering::SeqCst))
}
```

Try the demo

Shell:

```
cargo run
```

Browse http://localhost:3000/count or run:

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

You should see the hit count.

```
1
```

Reload and you should see the hit count increment:

You should see the hit count.

```
2
```

Create axum routes and axum handlers

An axum “route” is how you declare you want to receive a specific inbound HTTP requests; then you send the request to an axum “handler” function to do the work and return a response.

This section shows how to:

- Respond with a string of HTML
- Respond with a file of HTML
- Respond with HTTP status code
- Respond with the request URI
- Respond with a custom header and a custom image
- Respond with a JSON payload
- Respond to multiple HTTP verbs

Respond with string of HTML

Edit file `main.rs`.

Add a route:

```
let app = axum::Router::new()…

    .route("/string.html",
        get(string_html)
    )…
```

Add a handler:

```
/// axum handler for "GET /string.html" which responds with a string.
/// The `Html` type sets an HTTP header content-type of `text/html`.
pub async fn string_html() -> axum::response::Html<&'static str> {
    "<html><body><h1>Headline</h1><p>Paragraph</b></body></html>".into()
}
```

Try the demo

Shell:

```
cargo run
```

Browse http://localhost:3000/string.html or run:

```
curl 'http://localhost:3000/string.thml'
```

You should see the headline “Headline” and paragraph “Paragraph”.

Extractors

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 JSON parameters

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("/demo-path/{id}",
        get(get_demo_path_id)
    );
```

Add a handler:

```
/// axum handler for "GET /demo-path/{id}" which uses `axum::extract::Path
/// This extracts a path parameter then deserializes it as needed.
pub async fn get_demo_path_id(
    axum::extract::Path(id):
        axum::extract::Path<String>
) -> String {
    format!("Get demo path id: {:?}", id)
}
```

Try the demo

Shell:

```
cargo run
```

Shell:

```
curl 'http://localhost:3000/demo-path/1'
```

Output:

```
Get demo path 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 JSON parameters

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 `Cargo.toml` to add dependencies:

```
serde = { version = "~1.0.219", features = ["derive"] } # A serialization/
serde_json = { version = "~1.0.140" } # Serde serialization/deserializatio
```

Edit file `main.rs`.

Add a route to put the demo JSON:

```
let app = Router::new()…

    .route("/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")})
```

-

More about axum

This section covers three topics that can help you with specific needs for axum.

- RESTful routes and resources
- Tracing subscriber
- Bind + host + port + socket address

RESTful routes and resources

This section demonstrates how to:

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

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 Lazy for creating a global variable e.g. our DATA.
use std::sync::LazyLock;

/// 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: LazyLock<Mutex<HashMap<u32, Book>>> = LazyLock::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>
```

Post a new book

Edit file `main.rs`.

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

```
let app = Router::new()…

    .route("/books",
        get(get_books)
        .post(post_books)
    );
```

Add a handler:

```
/// axum handler for "POST /books" which creates a new book resource.
/// This demo shows how axum can extract JSON data into a Book struct.
pub async fn post_books(
    axum::extract::Json(book): axum::extract::Json<Book>
) -> axum::response::Html<String> {
    thread::spawn(move || {
        let mut data = DATA.lock().unwrap();
        let id = data.keys().max().unwrap() + 1;
        let book = Book { id, ..book };
        data.insert(id, book.clone());
        format!("Post a new book with new id {}: {}", &id, &book)
    }).join().unwrap().into()
}
```

Try the demo

Shell:

```
cargo run
```

Shell:

```
curl \
--request POST 'http://localhost:3000/books/0' \
--header "Content-Type: application/json" \
--data '{"id":0,"title":"Decameron","author":"Giovanni Boccaccio"}'
```

Output:

```
Post a new book with new id 4: 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

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 HT
/// This demo app uses our DATA variable, and iterates on it to find the t
pub async fn get_books_id(
    axum::extract::Path(id): axum::extract::Path<u32>
) -> axum::response::Html<String> {
    thread::spawn(move || {
        let data = DATA.lock().unwrap();
        match data.get(&id) {
            Some(book) => format!("<p>{}</p>\n", &book),
            None => format!("<p>Book id {} not found</p>", id),
        }
    }).join().unwrap().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/{id}` to append the function `put`:

```
let app = Router::new()…

    .route("/books/{id}",
        get(get_books_id)
        .put(put_books_id)
    );
```

Add a handler:

```
/// axum handler for "PUT /books/{id}" which sets a specific book resource
/// This demo shows how axum can extract JSON data into a Book struct.
pub async fn put_books_id(
    axum::extract::Json(book): axum::extract::Json<Book>
) -> axum::response::Html<String> {
    thread::spawn(move || {
        let mut data = DATA.lock().unwrap();
        data.insert(book.id, book.clone());
        format!("Put book: {}", &book)
    }).join().unwrap().into()
}
```

Try the demo

Shell:

```
cargo run
```

Shell:

```
curl 'http://localhost:3000/books/5'
```

Output:

```
<p>Book id 5 not found</p>
```

Shell:

```
curl \
--request PUT 'http://localhost:3000/books/4' \
--header "Content-Type: application/json" \
--data '{"id":5,"title":"Emma","author":"Jane Austen"}'
```

Output:

```
Put book: Emma by Jane Austen
```

Shell:

```
curl 'http://localhost:3000/books/5'
```

Output:

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

```
Delete book id: 1
```

Shell:

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

Output:

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

Patch one book

Create file `book_patch.rs`.

Add code for a BookPatch struct that has optional attributes:

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

// Demo book patch structure with some example fields for id, title, autho
// A production app could prefer an id to be type u32, UUID, etc.
#[derive(Debug, Deserialize, Clone, Eq, Hash, PartialEq)]
pub struct BookPatch {
    pub id: u32,
    pub title: Option<String>,
    pub author: Option<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 BookPatch {
    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 use the new BookPatch:

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

Modify the route `/books/{id}` to append the function `patch`:

```
let app = Router::new()…

    .route("/books/{id}",
        get(get_books_id)
        .put(put_books_id)
        .patch(patch_books_id)
    );
```

Add a handler:

```
/// axum handler for "PATCH /books/{id}" which updates attributes.
/// This demo shows how to mutate the book attributes in the DATA store.
pub async fn patch_books_id(
    axum::extract::Json(book_patch): axum::extract::Json<BookPatch>
) -> axum::response::Html<String> {
    thread::spawn(move || {
        let id = book_patch.id;
        let mut data = DATA.lock().unwrap();
        if data.contains_key(&id) {
            if let Some(title) = book_patch.title {
                data.get_mut(&id).unwrap().title = title.clone();
            }
            if let Some(author) = book_patch.author {
                data.get_mut(&id).unwrap().title = author.clone();
            }
            format!("Patch book id: {}", &id)
        } else {
            format!("Book id not found: {}", &id)
        }
    }).join().unwrap().into()
}
```

Try the demo

Shell:

```
cargo run
```

Shell:

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

Output:

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

Shell:

```
curl \
--request PATCH 'http://localhost:3000/books/1' \
--header "Content-Type: application/json" \
--data '{"id":1,"title":"Elektra"}'
```

Shell:

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

Output:

```
<p>Elektra by Sophocles</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> {
    thread::spawn(move || {
        let data = DATA.lock().unwrap();
        match data.get(&id) {
            Some(book) => format!(
                concat!(
                    "<form method=\"patch\" 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),
        }
    }).join().unwrap().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>
```

Patch 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)
        .patch(patch_books_id_form)
    );
```

Add a handler:

```
/// axum handler for "PATCH /books/{id}/form" which submits an HTML form.
/// This demo shows how to do a form submission then patch a resource.
pub async fn patch_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) {
            if !new_book.title.is_empty() {
                data.get_mut(&new_book.id).unwrap().title = new_book.title
            }
            if !new_book.author.is_empty() {
                data.get_mut(&new_book.id).unwrap().author = new_book.auth
            }
            format!("Patch book: {}", &new_book)
        } else {
            format!("Book id not found: {}", &new_book.id)
        }
    }).join().unwrap().into()
}
```

Try the demo

Shell:

```
cargo run
```

Shell:

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

Output:

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

Shell:

```
curl \
--request PATCH 'http://localhost:3000/books/1/edit' \
--header "Content-Type: application/x-www-form-urlencoded" \
--data 'id=1&title=Elektra'
```

Shell:

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

Output:

```
<p>Elektra by Sophocles</p>
```

-

Tracing subscriber

Edit file `Cargo.toml`.

Add dependencies:

```
tracing = { version = "~0.1.41" } #  Application-level tracing for Rust.
tracing-subscriber = { version = "~0.3.19", features = ["env-filter"] } #
```

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 to the start of the main function:

```
#[tokio::main]
pub async fn main() {
    // Start tracing and emit a tracing event.
    tracing_subscriber::registry()
        .with(tracing_subscriber::fmt::layer())
        .init();
    tracing::event!(tracing::Level::INFO, "main");
```

Try the demo

Shell:

```
cargo run
```

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

```
2024-08-31T20:06:34.894986Z  INFO demo_rust_axum: tracing
```

Bind & host & port & socket address

To bind the server listener, one way is to specify the host and port as a string such as:

```
let listener = tokio::net::TcpListener::bind("0.0.0.0:3000").await.unwrap
```

If you prefer to bind by using a host variable, a port variable, and a socket address variable, then you can edit file `main.rs`.

Modify the listener code to use whatever host and port and socket address you want:

```
let host = [127, 0, 0, 1];
let port = 3000;
let addr = std::net::SocketAddr::from((host, port));
let listener = tokio::net::TcpListener::bind(addr).await.unwrap();
```

Try the demo

Shell:

```
cargo run
```

Browse to your preferred host and port or run curl.

You should see typical console output.

axum repository examples

The axum 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

Ideas for next steps

- axum_session_auth
- loco.rs
- axum-layout
- aide: Open API documentation generator library.
- schemars: Generate JSON Schema documents from Rust code.
- plotars: Visualize data using Plotly and Polars.
- Utopia-axum: Bindings for Axum and utoipa (Simple, Fast, Code first and Compile time generated OpenAPI documentation for Rust).

Conclusion

You learned how to:

- Create a project using Rust and the axum web framework.
- Launch an axum server and run it with graceful shutdown.
- 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 axum 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.com

https://linkedin.com/in/joelparkerhenderson

https://github.com/joelparkerhenderson