



# Introduction to Axum: Building Web Servers in Rust

Welcome to the workshop on building web servers with **Axum**! In this session, we'll explore why Rust and Axum are a powerful combination for modern web development. Let's dive in!

## What is Axum?

**Axum** is a fast, ergonomic, and type-safe web framework for Rust, built on top of the `tokio` async runtime and `hyper` HTTP library. Developed by the Tokio team, Axum leverages Rust's strengths to provide a robust foundation for building scalable and reliable web services.

## Key Features and benefits

### 1. Declarative routing:

- define routes and handlers using a clean, intuitive API
- Supports RESTful patterns, path parameters, and query extraction

### 2. Middleware and composability:

- Integrates seamlessly with `tower` middleware (e.g., logging, CORS, rate limiting, observability).
- Middleware can be applied globally, per route, or per scope.

### 3. Async-first:

- Built on top of `tokio`, Axum is designed for high-performance, non-blocking I/O.
- Write handlers using `async/await` for efficient request processing.

### 4. JSON and Form handling:

- Effortless serialization/deserialization using `serde`.
- Extract and validate data with types like `Json<T>` or `Form<T>`.

## Why Rust for Web servers?

### Performance

- Rust compiles to native code, offering C-level speed with no garbage collector overhead.
- Ideal for high-throughput, low-latency applications.
- No runtime overhead thanks to Rust's zero-cost abstractions.

### Safety

- Rust's type system ensures correctness at compile time (e.g., path parameters, request bodies).
- Rust's ownership model eliminates entire classes of bugs (e.g., null pointers, data races).
- Critical for security in networked services.

### Concurrency Without Fear

- `Async/await` syntax simplifies writing safe, concurrent code.
- Avoid callback hell or runtime magic seen in other languages.

### Growing Ecosystem

- A vibrant community with libraries for HTTP, databases, auth, and more.
- Tools like `cargo` make dependency management painless.

### Cross-Platform Support

- Compile to Linux, Windows, macOS, or even embedded systems.

## Why Axum over other frameworks?

1. **Batteries-included, but unopinionated:** Axum provides essential tools without forcing a specific project structure.
2. **Production ready:** Backed by `tokio` and `hyper`, it powers high-traffic services in production.
3. **Future-proof:** Rust's stability guarantees ensure long-term maintainability.
4. **Learning Curve Pays Off:** While Rust has a steeper initial learning curve, the long-term benefits in reliability and performance are unmatched.

## Use cases

- Building microservices or REST/GraphQL APIs.
- High-performance proxies or gateways.
- Applications where safety and correctness are critical (e.g., fintech, healthcare).

 [Edit this page](#)

# Prerequisites

Before you start building a web server service with Axum, you need to have Rust installed on your system.

It's advised to use the latest stable version of Rust for this workshop.

## Installation guide

### macOS, Linux, or another Unix-like OS.

1. Open a terminal
2. Run the following command:

```
curl --proto 'https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

3. Follow the prompts to complete the installation.
4. Restart your terminal after installation.

### Windows

- **Website:** <https://www.rust-lang.org/>
- **Steps:**
  - i. Go to the Rust website.
  - ii. Click the **"Get Started"** button.
  - iii. Click the **"Other Installation Methods"** hyperlink
  - iv. Click on `rustup-init.exe` to get the installer.
  - v. Run the installer and follow the prompts.
    - When prompted, choose **"Proceed with installation"**.
  - vi. Restart your computer after installation.

## Verify Installation

After installing the prerequisites, verify your setup by running the following commands in your terminal:

```
rustc --version
cargo --version
```

## Troubleshooting

### 1. Rust/Cargo Errors

- Update Rust:

```
rustup update
```
- Visit <https://www.rust-lang.org/> and reinstall Rust if needed.

[✎ Edit this page](#)

# Getting started

Before starting building your HTTP server, ensure your system meets the requirements from the [Prerequisites](#) section.

## Scaffold your project

1. Open a terminal
2. Run the following command:

```
mkdir upb-rust-workshop && cd upb-rust-workshop
cargo init
```

### Project structure

```
upb-rust-workshop/
├─ src/
│   └─ main.rs      # Entry point
├─ target/          # Build artifacts
├─ .gitignore       # Git ignore file
├─ Cargo.lock       # Dependency lock file
└─ Cargo.toml       # Project manifest
```

## Add dependencies

Run the following commands:

```
cargo add axum
cargo add tokio --features=full
```

Next, check that your `Cargo.toml` file looks like this:

```
Cargo.toml

[package]
name = "upb-rust-workshop"
version = "0.1.0"
edition = "2024"

[dependencies]
axum = "0.8.1"
tokio = { version = "1.44.1", features = ["full"] }
```

name, version, edition and dependencies versions may vary.

## Scaffold your server

Open the `src/main.rs` file and add the following code:

```
src/main.rs

use axum::routing::get;

#[tokio::main]
async fn main() {
    let app = axum::Router::new().route("/", get(async || "Hello, World!"));

    let listener = tokio::net::TcpListener::bind("0.0.0.0:8080").await.unwrap();
    axum::serve(listener, app).await.unwrap();
}
```

## Run in development mode

Run the following command:

```
cargo run
```

Open your browser and navigate to <http://localhost:8080>. You should see the message `Hello, World!`.

## Build and run

To build and run your server in release mode, run the following command:

```
cargo build --release
```

Then, run the binary:

```
./target/release/upb-rust-workshop
```

Open your browser and navigate to <http://localhost:8080>. You should see the message `Hello, World!`.

[✎ Edit this page](#)

# Build a system monitor

Welcome to the workshop on building a system monitor with **Axum**! In this session, we'll explore how to create a high-performance HTTP server that exposes real-time and on-demand metrics about the system it's running on—perfect for monitoring resource usage, debugging, or integrating with observability tools.

## What You'll Build

Your system monitor server will expose four key endpoints:

1. `GET /healthcheck`: A simple endpoint to verify the server is running.
2. `GET /metrics`: Retrieve a comprehensive summary of system metrics.
3. `GET /metrics/{kind}`: Fetch specific metrics (e.g., system, process, memory, cpu, or disk) to drill down into performance data.
4. `GET /realtime`: Stream live metric updates using Server-Sent Events (SSE), providing a real-time dashboard experience.

## Why This Project?

By building this server, you'll gain practical experience with:

- **Rust's async ecosystem**: Using Axum and Tokio to handle concurrent HTTP requests.
- **System programming**: Interacting with low-level OS APIs to collect metrics like CPU usage, memory allocation, and disk I/O.
- **Real-time communication**: Implementing SSE to push updates to clients without polling.
- **API design**: Structuring clean, maintainable endpoints for extensibility.

Whether you're new to systems programming or looking to deepen your Rust expertise, this project bridges the gap between theory and real-world applications. Let's dive in and build a tool that's as educational as it is practical! 🚀

## Skill Progression

1. **Task 1**: Organize the project structure and set up the `/healthcheck` endpoint.
2. **Task 2**: Implement the `/metrics` and `/metrics/{kind}` endpoints.
3. **Task 3**: Add the `/realtime` endpoint to server realtime metrics.

[✏️ Edit this page](#)

# Task 1: Organize the project structure and set up the `/healthcheck` endpoint

Before diving in, first we'll setup some conventions for organizing the code, which will help us maintain a clean and scalable project structure, benefits of the Rust module system.

The router will live inside the `src/routes` module, and each endpoint will have its own module. This way, we can keep the codebase organized and easy to navigate.

To do so, we'll update the directory structure as such:

```
upb-rust-workshop/
├── src/
│   ├── routes/
│   │   └── mod.rs
│   └── main.rs # Entry point
├── target/     # Build artifacts
├── .gitignore  # Git ignore file
├── Cargo.lock  # Dependency lock file
└── Cargo.toml  # Project manifest
```

Within `src/routes/mod.rs`, we'll move the router initialization from `main.rs` as follows:

```
src/routes/mod.rs

use axum::Router;

pub fn app() -> Router {
    Router::new().route("/", get(async || "Hello, World!"))
}
```

After that, the `main.rs` will become:

```
src/main.rs

pub mod routes;

use axum;

#[tokio::main]
async fn main() {
    let listener = tokio::net::TcpListener::bind("0.0.0.0:8080").await.unwrap();
    println!(
        "Server running on http://{}",
        listener.local_addr().unwrap()
    );
    axum::serve(listener, routes::app()).await.unwrap();
}
```

If you run the server now, you should see the message `Server running on http://0.0.0.0:8080`, and the `/` endpoint should return `Hello, World!`.

Now, let's add a `/healthcheck` endpoint to our server. This endpoint will be used to check if the server is running and healthy.

To do so, create a new module `src/routes/healthcheck.rs` and add the following code:

```
src/routes/healthcheck.rs

use axum::{Router, http::StatusCode, response::IntoResponse, routing::get};

pub fn register() -> Router {
    Router::new().route("/", get(health_check))
}

async fn health_check() -> impl IntoResponse {
    todo!("Implement the health check endpoint")
}
```

Next, update `src/routes/mod.rs` to include the new module:

```
src/routes/mod.rs

use axum::Router;

mod healthcheck;

pub fn app() -> Router {
    Router::new().nest("/healthcheck", healthcheck::register())
}
```

Why we did that? By nesting the `/healthcheck` route inside the main router, we can easily add more routes in the future without cluttering the main router.

## Your Task

As task, implement the `health_check` function to return a `200 OK` status code with the message `Server is running`.

## Conclusion

In this task, we've organized the project structure and added a `/healthcheck` endpoint to our server. This will help us maintain a clean and scalable codebase as we add more features to our web server.

[✎ Edit this page](#)

## Task 2: Implement the `/metrics` and `/metrics/{kind}` endpoints

For checking the system metrics, we'll need a little design before. So, we want to get all the metrics at once, and we also want to get specific metrics. For this, we'll have two endpoints:

1. `GET /metrics`: Retrieve a comprehensive summary of system metrics.
2. `GET /metrics/{kind}`: Fetch specific metrics (e.g., system, process, memory, cpu, or disk) to drill down into performance data.

To register the new endpoints, follow the steps below:

1. create a new module `src/routes/metrics.rs` and add the following code:

```
src/routes/metrics.rs

use axum::{Router, extract::Path, http::StatusCode, response::IntoResponse, routing::get};

pub fn register() -> Router {
    Router::new()
        .route("/", get(get_metrics))
        .route("/{kind}", get(get_metric))
}

async fn get_metrics() -> impl IntoResponse {
    todo!("Implement the get_metrics endpoint")
}

async fn get_metric(kind: /* TODO */ -> impl IntoResponse {
    todo!("Implement the get_metric endpoint")
}
```

2. Update `src/routes/mod.rs` to include the new module:

```
src/routes/mod.rs

use axum::Router;

mod healthcheck;
mod metrics;

pub fn app() -> Router {
    Router::new()
        .nest("/healthcheck", healthcheck::register())
        .nest("/metrics", metrics::register())
}
```

Also, we should restrict the values that `kind` can take. To do so, we will represent them within the type system as an enum, like such:

1. Create a new module `src/metrics.rs`.
2. Add the following code to the new file:

```
src/metrics.rs

pub enum Kind {
    System,
    Process,
    Memory,
    Cpu,
    Disk,
}
```

3. Add the following line to `main.rs`:

```
src/main.rs

mod metrics;
```

4. Finally, we'll update the `get_metric` handler to take a `Kind` parameter:

```
src/routes/metrics.rs

// ...

use crate::metrics;

async fn get_metric(Path(kind): Path<metrics::Kind>) -> impl IntoResponse {
    todo!("Implement the get_metric endpoint")
}
```

`Path` is a type provided by Axum that allows you to extract a part of the request path. In this case, we're using it to extract the `kind` parameter from the request path.

Ok so we designed the request format and the endpoints, now, let's implement the logic to get the metrics.

I've created some helper functions in `src/metrics.rs` to get the system metrics. You can use them to implement the `/metrics` and `/metrics/{kind}` endpoints.

```
src/metrics.rs

use sysinfo;

pub async fn init() -> sysinfo::System {
    let mut sys = sysinfo::System::new_all();
    sys.refresh_all();

    tokio::time::sleep(sysinfo::MINIMUM_CPU_UPDATE_INTERVAL).await;
    sys.refresh_cpu_all();

    sys
}

pub enum Kind {
    System,
```

```

    Process,
    Memory,
    Cpu,
    Disk,
}

pub struct System {
    name: String,
    kernel_version: String,
    os_version: String,
    host_name: String,
    uptime: u64,
}

impl System {
    pub fn generate() -> Self {
        todo!("Implement the System::generate method")
    }
}

pub struct Process {
    pid: u32,
    name: String,
    memory: u64,
    cpu_usage: f32,
    run_time: u64,
}

impl Process {
    pub fn generate(sys: &mut sysinfo::System) -> Vec<Self> {
        todo!("Implement the Process::generate method")
    }
}

#[derive(serde::Serialize, serde::Deserialize)]
pub struct Memory {
    used: u64,
    total: u64,
}

impl Memory {
    pub fn generate(sys: &mut sysinfo::System) -> Self {
        todo!("Implement the Memory::generate method")
    }
}

#[derive(serde::Serialize, serde::Deserialize)]
pub struct CoreMetrics {
    name: String,
    brand: String,
    usage: f32,
    frequency: u64,
}

#[derive(serde::Serialize, serde::Deserialize)]
pub struct Cpu {
    cpu_usage: f32,
    cores: Vec<CoreMetrics>,
}

impl Cpu {
    pub fn generate(sys: &mut sysinfo::System) -> Self {
        todo!("Implement the Cpu::generate method")
    }
}

#[derive(serde::Serialize, serde::Deserialize)]
pub struct Disk {
    name: String,
    available_space: u64,
    total_space: u64,
    is_removable: bool,
}

impl Disk {
    pub fn generate() -> Vec<Self> {
        todo!("Implement the Disk::generate method")
    }
}

#[derive(serde::Serialize, serde::Deserialize)]
pub struct Summary {
    system: System,
    process: Vec<Process>,
    memory: Memory,
    cpu: Cpu,
    disk: Vec<Disk>,
}

impl Summary {
    pub fn generate(sys: &mut sysinfo::System) -> Self {
        todo!("Implement the Summary::generate method")
    }
}

```

To get the system metrics, we're using the `sysinfo` crate. It provides a simple interface to get system information like CPU usage, memory usage, disk space, etc. You can find more information about the crate [here](#). Install the crate similar to how you installed Axum and Tokio.

## What is serde?

`serde` is a popular Rust library for serializing and deserializing data. It provides a simple way to convert Rust data structures into JSON, XML, or other formats. In this project, we're using `serde` to serialize our metrics into JSON format.

The name `serde` comes from "serialization" and "deserialization."

Because our HTTP server uses JSON as the default format for responses, we need to implement the `serde::Serialize` trait for our metric structs. This trait allows us to convert our structs into JSON objects that can be sent over the network.

To add `serde` to our project, run the following commands in a terminal:

```

cargo add serde --features derive
cargo add serde_json

```

To convert a struct into JSON, you can use the `serde_json::to_string` function. For example:

```

serde_json::to_string(&my_struct).unwrap()

```


## Your task

As task, implement the `get_metrics` and `get_metric` functions to return a `200 OK` status code with the system metrics. The `get_metrics` function should return a summary of all the metrics, while the `get_metric` function should return the specific metric based on the `kind` parameter.

Also, implement the `generate` methods for the `System`, `Process`, `Memory`, `Cpu`, `Disk`, and `Summary` structs to generate the metrics.

## Conclusion

In this task, we've designed the `/metrics` and `/metrics/{kind}` endpoints and implemented the logic to get the system metrics. By following the steps above, you've learned how to structure your project, define routes, and handle requests in Axum. Next, we'll add the `/realtime` endpoint to stream live metric updates using Server-Sent Events (SSE). Let's continue building our system monitor server! 🚀

 [Edit this page](#)



[🏠](#) [Axum Web Servers](#) [Build a system monitor](#) [Task 3 - Make metrics realtime](#)

## Task 3: Add the `/realtime` endpoint to server realtime metrics.

Last time, we added the `/metrics` and `/metrics/{kind}` endpoints to our server. Now, a natural feature we could implement is a real-time endpoint that sends metric updates at a fixed rate. This will allow clients to subscribe to the endpoint and receive live updates without polling the server.

To do so, we'll implement the `/realtime` endpoint using Server-Sent Events (SSE). This technology allows servers to push updates to clients over a single, long-lived connection.

### What are Server-Sent Events (SSE)?

Server-Sent Events (SSE) is a standard for sending real-time updates from a server to a client over HTTP. It's a simple and efficient way to stream data from the server to the client without the need for polling.

SSE works by establishing a persistent connection between the client and the server. The server can then send messages to the client at any time, and the client will receive them as they arrive.

In our project, we'll use SSE to stream live metric updates to clients who connect to the `/realtime` endpoint.

### Your Task

As task, implement the `/realtime` endpoint to stream live metric updates using Server-Sent Events (SSE). The endpoint should send a message every second with the current system metrics.

To do so, we'll use the `tokio-stream` crate to create a stream that sends updates at a fixed rate. We'll then use the `axum::sse` module to send these updates to clients.

### Hint

To generate a new metric every second, we'll use the following code:

```
use std::time::Duration;
use tokio_stream::{StreamExt, wrappers::IntervalStream};

let stream = IntervalStream::new(tokio::time::interval(Duration::from_secs(1)));
```

[✎ Edit this page](#)

# Links and docs

- [Rust](#)
- [axum](#)
- [tokio](#)
- [hyper](#)
- [sysinfo](#)
- [serde](#)

 [Edit this page](#)