Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

HTTP e REST

A forma mais simples de criar servidores **HTTP/REST** em **Rust** é utilizar o **tiny-http**, que é o que usaremos aqui neste curso. O objetivo é mostrar o básico de rust, mas faremos outros cursos mais avançados.

Para criar **serviços RESTful** em Rust usando **tiny_http**, basta seguir alguns passos conceituais, sem precisar de muitos recursos avançados:

1. **Criar o projeto**
   Inicie um novo projeto Rust com `cargo new`, o que estabelece a estrutura básica (arquivo `Cargo.toml` e diretório `src`).

2. **Adicionar tiny_http às dependências**
   No `Cargo.toml`, inclua a dependência `tiny_http`, que gerencia a parte fundamental de um servidor HTTP simples (abertura de porta, parsing das requisições básicas, envio de respostas).

3. **Inicializar o servidor**
   No código principal, você utiliza o `tiny_http::Server` para escutar em um endereço específico (por exemplo, `127.0.0.1:3030`). Esse server tem um loop principal que aceita conexões e produz `Request` para cada requisição recebida.

4. **Processar as requisições**
   Dentro do loop de conexão, você lê cada `Request` e decide o que fazer com base em:

   - **Método** (GET, POST, etc.)
   - **URL** (ex.: `/users`, `/ping` etc.)
   - **Cabeçalhos**, se necessário (por exemplo, para checar `Authorization` ou `Content-Type`).

5. **Definir rotas**
   Cada combinação de método e caminho (por exemplo, `GET /ping`, `POST /users`) representa uma "rota" da sua API. Você pode escolher usar `match` ou `if/else` para direcionar a requisição para a lógica correspondente.

6. **Tratar dados de entrada**
   Em caso de requisições como POST ou PUT, você costuma ler o corpo da mensagem (`request.as_reader()`), que normalmente está em JSON. Se quiser converter este JSON diretamente para structs Rust, é comum usar `serde` e `serde_json` para deserialização.

7. **Produzir respostas**
   Depois de processar a lógica (por exemplo, salvar dados, consultar alguma informação), você retorna um objeto `Response` do `tiny_http`. Para dados JSON, basta montar uma string contendo JSON e configurar o cabeçalho `Content-Type: application/json`.

8. **Implementar autenticação (opcional)**
   Se desejar proteção via JWT, você:

   - Cria uma rota de login que gera um token a partir de dados do usuário.
   - Lê o cabeçalho `Authorization` nas rotas que requerem autenticação, extrai o token e valida com a biblioteca `jsonwebtoken`.
   - Caso o token não seja válido ou ausente, retorna um status de erro (`401 Unauthorized`, por exemplo).

9. **Adicionar validações e status apropriados**
   Em uma API RESTful, cada rota deve retornar códigos de status coerentes (200 para sucesso, 404 para recursos inexistentes, 400 para problemas no body JSON, etc.).

10. **Manter o servidor em loop**
    O servidor `tiny_http` fica num loop infinito (até ser encerrado), aceitando requisições e chamando suas rotas. Esse modelo é bem simples e dá controle total sobre o que acontece a cada requisição.

Em resumo, **tiny_http** oferece só o essencial: abrir porta, receber requisições e enviar respostas em formato HTTP. Você mesmo gerencia roteamento, parse de JSON e a lógica da aplicação, o que o torna uma escolha bem minimalista para criar serviços RESTful em Rust.

Um exemplo simples

Na pasta de código do curso há uma subpasta "uuid_gen" que contém um RESTful service capaz de gerar **UUIDs**. Veja o `Cargo.toml`:

```
[package]
name = "meu_uuid"
version = "0.1.0"
edition = "2021"

[dependencies]
tiny_http = "0.9"
uuid = { version = "1.3", features = ["v4"] }
```

```
serde_json = "1.0"
serde = { version = "1.0", features = ["derive"] }
```

Agora, veja o código-fonte:

```
use tiny_http::{Server, Response, Method, Header};
use serde_json::json;
use uuid::Uuid;

fn main() {
    // Inicia o servidor HTTP na porta 3030
    let server = Server::http("127.0.0.1:3030").unwrap();
    println!("Servidor rodando em http://127.0.0.1:3030");

    // Loop infinito para aceitar e processar requisições
    for request in server.incoming_requests() {
        match (request.method(), request.url()) {
            // Rota GET /uuid
            (&Method::Get, "/uuid") => {
                // Gera um novo UUID v4
                let new_uuid = Uuid::new_v4();

                // Constrói o JSON de resposta
                let response_body = serde_json::to_string(&json!({
                    "uuid": new_uuid.to_string()
                })).unwrap();

                // Monta e envia a resposta (status 200, content-type JSON)
                let response = Response::from_string(response_body)
                    .with_header(Header::from_bytes("Content-Type",
"application/json").unwrap())
                    .with_status_code(200);
                let _ = request.respond(response);
            },
            // Se não for /uuid, retorna 404
            _ => {
                let response = Response::from_string("Not
Found").with_status_code(404);
                let _ = request.respond(response);
            }
        }
    }
}
```

Execução do servidor:

```
$ cargo run
Servidor rodando em http://127.0.0.1:3030
...
```

```
$ curl http://localhost:3030/uuid
{"uuid":"7fb6ae6c-6eed-4fcc-99ed-abcc5c083b9b"}
```

Explicação do código

Vamos pontuar o que acontece em cada parte desse código para que você entenda a lógica por trás:

1. **Iniciando o servidor**

```
let server = Server::http("127.0.0.1:3030").unwrap();
```

   - O `tiny_http::Server::http(...)` abre um socket TCP em `127.0.0.1:3030`.
   - Caso ocorra algum erro (por exemplo, se a porta estiver ocupada), o `unwrap()` interrompe o programa.

2. **Loop principal de requisições**

```
for request in server.incoming_requests() {
    // ...
}
```

   - `incoming_requests()` produz um iterador de requisições que chegam ao servidor.
   - Cada iteração do `for` lida com uma única requisição HTTP.

3. **Tratando a requisição**

```
match (request.method(), request.url()) {
    (&Method::Get, "/uuid") => { ... }
    _ => { ... }
}
```

   - Verifica **qual** método (GET, POST, etc.) e **qual** URL (`/uuid`, `/ping`, etc.) chegaram.
   - Aqui, estamos lidando especificamente com `GET /uuid`. Qualquer outra rota cairá no `_ => { ... }` (ou seja, 404).

4. **Gerando UUID**

```
let new_uuid = Uuid::new_v4();
```

   - Gera um UUID v4 aleatório (ex.: `7a951b0f-dfaa-4171-9fcb-4381f6b1a964`).
   - Esse método funciona porque habilitamos a feature `"v4"` no `Cargo.toml` para a crate `uuid`.

5. **Montando JSON**

```
let response_body = serde_json::to_string(&json!({
    "uuid": new_uuid.to_string()
})).unwrap();
```

- `serde_json::json!` cria um objeto JSON rapidamente.
- `to_string` converte o objeto em uma string JSON, tipo `{"uuid":"7a951b0f-dfaa-4171-9fcb-4381f6b1a964"}`.
- O `unwrap()` interrompe o programa caso haja erro de conversão (improvável, mas é o jeito de simplificar a lógica).

6. **Criando a resposta**

```
let response = Response::from_string(response_body)
    .with_header(Header::from_bytes("Content-Type",
"application/json").unwrap())
    .with_status_code(200);
```

- `Response::from_string(...)`: cria um objeto de resposta com aquele corpo de texto.
- `.with_header(...)`: adiciona o cabeçalho `"Content-Type: application/json"` (indicando que o corpo é JSON).
- `.with_status_code(200)`: define o HTTP status como `200 OK`.

7. **Enviando ao cliente**

```
let _ = request.respond(response);
```

- `request.respond(...)` envia essa resposta através do socket para quem fez a requisição.
- O `let _ = ...;` é só para ignorar o `Result` retornado (se houve falha no envio, não tratamos).

8. **Qualquer outro caminho → 404**

```
_ => {
    let response = Response::from_string("Not Found")
        .with_status_code(404);
    let _ = request.respond(response);
}
```

- Se o método/rota não corresponder a `(&Method::Get, "/uuid")`, retorna um corpo simples `"Not Found"` com código `404`.

Em resumo:

- **tiny_http** gerencia a parte de “escutar a porta” e te entrega requisições.

- Você usa `match` no método e na URL para ver qual rota deve ser tratada.
- Para **GET /uuid**, gera um UUID (usando `uuid::Uuid`), transforma em JSON e retorna com cabeçalho `Content-Type: application/json`.
- Caso o caminho não seja conhecido, devolve um `404 - Not Found`.
- Tudo isso ocorre em um loop infinito que só termina quando o programa é interrompido.

O Tiny HTTP é Multithread?

tiny_http cria um thread interno (de "background") para **aceitar** conexões (isto é, para escutar na porta e aceitar novos sockets). Porém, quando falamos em "servidor HTTP multithread", normalmente estamos interessados em processar **Vamos esclarecer essa aparente contradição:

- **tiny_http** de fato cria **uma** thread interna (de "background") para **aceitar** conexões (isto é, para escutar na porta e aceitar novos sockets).
- Porém, quando falamos em "servidor HTTP multithread", normalmente estamos interessados em processar **várias requisições em paralelo**.

No caso do **tiny_http**, **por padrão**, se você fizer algo assim:

```
for request in server.incoming_requests() {
    // processa requisição
}
```

esse loop **não** está em múltiplas threads; apenas o accept de novas conexões (pegar o socket) acontece numa thread de fundo. O processamento de cada request (e a geração da resposta) ainda ocorre em **um só thread** (o seu loop).

Então tiny_http é single-thread ou multithread?

- Ele é **parcialmente** multithread: a parte de "aceitar conexões" acontece numa thread interna.
- **Porém**, ele **não** cria automaticamente um pool de threads para processar as requisições em paralelo.

Se você quer processar requisições **simultaneamente**, deve criar você mesmo várias threads que chamam `server.recv()` ou `server.incoming_requests()`. O `Server` é "clonável" (internamente ele é protegido por `Arc`), então você pode distribuir esse "handle" para várias threads, cada uma rodando o loop de requests. Exemplo simplificado:

```
use tiny_http::{Server, Response};
use std::sync::Arc;
use std::thread;

fn main() {
    let server = Arc::new(Server::http("0.0.0.0:3030").unwrap());

    // Exemplo: 4 threads para processar requisições
    for i in 0..4 {
        let server_clone = server.clone();
        thread::spawn(move || {
```

```
            println!("Thread {} iniciada", i);
            for request in server_clone.incoming_requests() {
                // Aqui cada thread processa as requisições que pegar
                let response = Response::from_string("Hello from
multithread!");
                let _ = request.respond(response);
            }
        });
    }

    // Impede a main de terminar (qualquer modo de "aguardar" serve)
    loop {
        std::thread::park();
    }
}
```

Nesse modelo, cada thread fica chamando `incoming_requests()` (ou `recv()`) em loop. Sempre que chega uma nova conexão, a thread interna do tiny_http aceita a conexão e coloca os requests numa fila interna. Então **qualquer** das quatro threads que estiver disponível (chamando `incoming_requests()`) consegue buscar o próximo request da fila e processá-lo.

Conclusão

- **tiny_http** não é 100% single-thread: ele tem **1 thread para aceitar conexões**.
- Ele **não** possui, por padrão, um **pool** de threads para executar suas requisições simultaneamente.
- Para processar **requisições** em paralelo, **você** deve criar manualmente várias threads e, dentro delas, chamar `incoming_requests()`.

Dessa forma, você tem o controle total de quantas threads vão lidar com as requisições — ao invés de um framework que já gerencie automaticamente. várias requisições em paralelo**.

No caso do **tiny_http**, **por padrão**, se você fizer algo assim:

```
for request in server.incoming_requests() {
    // processa requisição
}
```

esse loop **não** está em múltiplas threads; apenas o accept de novas conexões (pegar o socket) acontece numa thread de fundo. O processamento de cada request (e a geração da resposta) ainda ocorre em **um só thread** (o seu loop).

Então tiny_http é single-thread ou multithread?

- Ele é **parcialmente** multithread: a parte de “aceitar conexões” acontece numa thread interna.
- **Porém**, ele **não** cria automaticamente um pool de threads para processar as requisições em paralelo.

Se você quer processar requisições **simultaneamente**, deve criar você mesmo várias threads que chamam `server.recv()` ou `server.incoming_requests()`. O `Server` é “clonável” (internamente ele é

protegido por `Arc`), então você pode distribuir esse "handle" para várias threads, cada uma rodando o loop de requests. Exemplo simplificado:

```
use tiny_http::{Server, Response};
use std::sync::Arc;
use std::thread;

fn main() {
    let server = Arc::new(Server::http("0.0.0.0:3030").unwrap());

    // Exemplo: 4 threads para processar requisições
    for i in 0..4 {
        let server_clone = server.clone();
        thread::spawn(move || {
            println!("Thread {} iniciada", i);
            for request in server_clone.incoming_requests() {
                // Aqui cada thread processa as requisições que pegar
                let response = Response::from_string("Hello from
multithread!");
                let _ = request.respond(response);
            }
        });
    }

    // Impede a main de terminar (qualquer modo de "aguardar" serve)
    loop {
        std::thread::park();
    }
}
```

Nesse modelo, cada thread fica chamando `incoming_requests()` (ou `recv()`) em loop. Sempre que chega uma nova conexão, a thread interna do tiny_http aceita a conexão e coloca os requests numa fila interna. Então **qualquer** das quatro threads que estiver disponível (chamando `incoming_requests()`) consegue buscar o próximo request da fila e processá-lo.

Então?

- **tiny_http** não é 100% single-thread: ele tem **1 thread para aceitar conexões**.
- Ele **não** possui, por padrão, um **pool** de threads para executar suas requisições simultaneamente.
- Para processar **requisições** em paralelo, **você** deve criar manualmente várias threads e, dentro delas, chamar `incoming_requests()`.

Dessa forma, você tem o controle total de quantas threads vão lidar com as requisições — ao invés de um framework que já gerencie automaticamente.

Exercício final

Utilizando a tabela `Pessoas`, utilizada na aula passada, crie um **RESTful** service que retorne as pessoas cadastradas na tabela. Use o **Diesel**. Ah, e o servidor tem que ser Multithread!

Não se apavore! Tudo o que necessita já foi providenciado, e a correção está na pasta de código. Mas tente fazer!

Crates populares para criar RESTful services:

Existem várias crates populares para criar serviços RESTful em Rust, entre as quais se destacam:

- **Actix-web**: extremamente rápido e robusto, costuma aparecer no topo de benchmarks.
- **Axum**: construída sobre `hyper` e `tower`, tem ganhado popularidade pela simplicidade e foco em ergonomia.
- **Warp**: funcional e minimalista, foco em filtros e combinadores.
- **Rocket**: intuitivo, com macros que simplificam rotas, porém precisa de recursos mais recentes do compilador.
- **Tide**: mais simples e assíncrono, criado pela equipe do Async-std.

Atualmente, **Actix-web** costuma ser visto como o mais popular e performático, mas **Axum** vem crescendo rapidamente em adoção.

Então você terminou?

Ótimo! Quer um certificado de conclusão? Ok. Agente uma entrevista comigo. Farei algumas perguntas e, se passar, emitirei um certificado assinado digitalmente para você. Isso tem custo. Envie email para: cleuton@cleutonsampaio.com.