**Rust to Robustness: Implementing a Reverse Proxy Server.**

[[](https://medium.com/@adefemiadeoye?source=post_page-----74f9f7e9604e--------------------------------)](https://medium.com/@adefemiadeoye?source=post_page-----74f9f7e9604e--------------------------------)

[Adeoye Adefemi](https://medium.com/@adefemiadeoye?source=post_page-----74f9f7e9604e--------------------------------)

·

Follow

3 min read

·

Sep 28

31



photo credit: Pexel

**Overview**

A proxy server is a service that routes requests from a client to another service.

Generally, proxy servers are classified into forward proxy servers and reverse proxy servers.

The following section would reflect on how I built a reverse proxy server in Rust, the motivation and the limitations.

**Motivation**

Let’s face it! Why do I need a reverse proxy server?

I’m building <https://github.com/opeolluwa/martus> a school management software (backend) as a pack of distributed services, managing different service URLs in the frontend would be a challenge

In addition, keeping track of the services the requests are sent is bad for developer experience.

The reverse proxy server would accept all requests and route them to the respective service, This will, among other things, improve the developer experience and it could be leveraged in strengthening the systems’s security.

**Choice of technologies**

The choice of technology (the Rust Programming language) is influenced by the need for a super fast application with minimal footprint on the computer on which it runs.

The reverse proxy server would accept all incoming requests, route them to the appropriate service, await response and return them to the clients.

**Implementation**

TL;DR!

The entire code for the reverse proxy server implementation can be found on <https://github.com/opeolluwa/martus-proxy-server>

I’d explain the system concepts herein rather than sharing the whole logic implementation.

The proxy server does 2 things.

First, It receives the HTTP request object and extract the parts required to make the transfer to the actual server, this includes: the**Request Headers,**the**Request Body,**the**HTTP Verb**(GET, POST, PATCH …), finally the **URL**

async fn handler(path: Uri, method: Method, headers: HeaderMap, body: Bytes) -> Response<Body> {  
 // pass data to request builder  
 let body = Body::from(body);  
 let path = path.path().split('/').collect::<Vec<&str>>();  
  
 ...  
}

Internally, it splits the URL, to extract the resource path. Suppose my proxy server runs on **http://0.0.0.0:5000**, the handler is configured to listen on route **http://0.0.0.0:5000/v1/\*path**

The pattern *\*path*used to allow the proxy server to receive a diverse form of request, finally, the \*path, is refined to identify the services the requests would be routed to, and optionally, the resource path.

For example in **http://0.0.0.0:5000/v1/admin/register**

The \*path is identical to **admin/register**

Similarly, in **http://0.0.0.0:5000/student/verify?token=eyJhbGciOi[…]nR5cCI6IkpXVCJ9,**

the pattern matches ***student/verify?token=eyJhbGciOi[…]nR5cCI6IkpXVCJ9***

Secondly, the matched pattern is split into 2, after the first slash to extract the service ID,( in this case “student” and “admin”), the trailing part of the pattern is the resource path.

Going further, the service ID is used to match the service URL to proxy the request to.

enum ServicePath<'a> {  
 Student(&'a str),  
 Admission(&'a str),  
 Library(&'a str),  
 Hostel(&'a str),  
}  
  
#[allow(dead\_code)]  
impl ServicePath<'\_> {  
 pub fn new(service\_path: &str) -> ServicePath<'\_> {  
 match service\_path {  
 "student" => ServicePath::Student("http://0.0.0.0:5001/"),  
 "admission" => ServicePath::Admission("http://0.0.0.0:5002/"),  
 "library" => ServicePath::Library("http://0.0.0.0:5003/"),  
 "hostel" => ServicePath::Hostel("http://0.0.0.0:5004/"),  
 \_ => ServicePath::Student("http://0.0.0.0:5000/"),  
 }  
 }  
  
 // read the url from env  
 fn from\_env<'a>(key: &'a str, default: &'a str) -> std::string::String {  
 std::env::var(key).unwrap\_or(default.to\_string())  
 }  
 // build a request url  
 pub fn build\_url(&self, resource\_path: &str) -> String {  
 let base\_url = match self {  
 ServicePath::Student(base\_url) => base\_url,  
 ServicePath::Admission(base\_url) => base\_url,  
 ServicePath::Library(base\_url) => base\_url,  
 ServicePath::Hostel(base\_url) => base\_url,  
 };  
 format!("{}{}", base\_url, resource\_path)  
 }  
}

Finally, a new http request is constructed and forwarded to the service, the proxy server awaits the response and returns it to the client

**Limitations**

The first limitation in the implementation is static binding of the service ID

impl ServicePath<'\_> {  
 pub fn new(service\_path: &str) -> ServicePath<'\_> {  
 match service\_path {  
 "student" => ServicePath::Student("http://0.0.0.0:5001/"),  
 "admission" => ServicePath::Admission("http://0.0.0.0:5002/"),  
 "library" => ServicePath::Library("http://0.0.0.0:5003/"),  
 "hostel" => ServicePath::Hostel("http://0.0.0.0:5004/"),  
 \_ => ServicePath::Student("http://0.0.0.0:5000/"),  
 }  
 }

This would imply I need to update the services URL from time to time. This could be improved on by using a configuration file or environment variables. Doing this would allow more use cases and would also make the services usable as an open source software for other people.

**Conclusion**

Building the proxy server is the first step to implementing a school management software, as I go on there would be issues and need to review the logic.