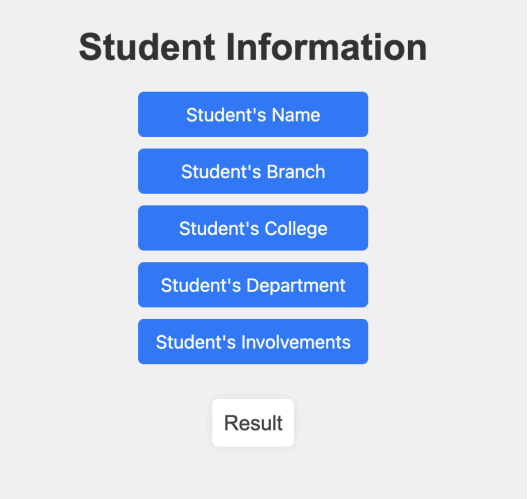
Documentation of Rust Web Server Task

By Divyansh Joshi

Basic Idea of the Project:

I have made a basic student information dashboard where there are multiple buttons and on clicking each button, it sends a request to port 8000 where I have hosted the backend. On clicking the corresponding button, a request is sent to the port 8000 which then returns the string which matches the query.



(*Frontend Interface*)

client > index.html > html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Information</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>Student Information</h1>
    <button id="button-1">Student's Name</button>
    <button id="button-2">Student's Branch</button>
    <button id="button-3">Student's College</button>
    <button id="button-4">Student's Department</button>
    <button id="button-5">Student's Involvements</button>
    <p id="result-paragraph">Result</p>
    <script src="script.js"></script>
</body>
</html>
```

(index.html)

It displays the buttons on the frontend and each button has the property it returns.

client > JS script.js > ...

```js
document.addEventListener("DOMContentLoaded", () => {
    const resultParagraph = document.getElementById("result-paragraph");

    function fetchData(endpoint) {
        fetch(`http://localhost:8000${endpoint}`)
            .then(res => {
                if (!res.ok) {
                    throw new Error(`HTTP error! Status: ${res.status}`);
                }
                return res.text();
            })
            .then(data => {
                console.log(data);
                resultParagraph.innerHTML = data;
            })
            .catch(err => {
                console.error(err);
                resultParagraph.innerHTML = "Error occurred";
            });
    }

    document.getElementById("button-1").addEventListener("click", () => fetchData("/name"));
    document.getElementById("button-2").addEventListener("click", () => fetchData("/branch"));
    document.getElementById("button-3").addEventListener("click", () => fetchData("/college"));
    document.getElementById("button-4").addEventListener("click", () => fetchData("/department"));
    document.getElementById("button-5").addEventListener("click", () => fetchData("/involvements"));
});

```

(script.js)

This has the function *fetchData* which sends a fetch request to the *localhost:8000* according to the button clicked.

The code of rust has been divided into two sub files : *main.rs* and *lib.rs* for better readability, structuring and reuse.

The *main.rs* is the executable binary crate whereas *lib.rs* is the library crate, which contains the functions that are used by our binary crate.

If a single thread would have been used, it would execute only one task at a time so the tasks would have to wait if there were multiple requests sent to the server. A multithreaded server, on the other hand, processes several requests concurrently, therefore if multiple requests will be sent to the server, the requests that came later will not have to wait when the earlier requests are being processed.

We cannot use a separate thread for each task as it would create a huge load on our system and it might crash if there are a lot of incoming requests to the server. The web server can handle 4 incoming requests at a time.

The format of a HTML request is :

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

The format of a response is :

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

The server uses a *ThreadPool* which basically is a pool of threads i.e. it has several threads. Whenever a request comes, a thread is assigned to perform that task and whenever it has performed the task, the thread is returned again in the pool for re-use.

server > src > main.rs > handle_connection

```rust
fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&stream);
    let request_lines: Vec<String> = buf_reader.lines()
        .map(|line| line.unwrap_or_default())
        .take_while(|line| !line.is_empty())
        .collect();

    if request_lines.is_empty() {
        return;
    }

    let request_line = &request_lines[0];
    println!("Received request: {}", request_line);

    let response = match request_lines[0].trim() {
        "GET / HTTP/1.1" => response_text("Click on buttons to get responses"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            response_text("Responded after delay")
        },
        "GET /name HTTP/1.1" => response_text("Divyansh Joshi"),
        "GET /branch HTTP/1.1" => response_text("Computer Science and Engineering"),
        "GET /college HTTP/1.1" => response_text("IIT Kharagpur"),
        "GET /involvements HTTP/1.1" => response_text("Communique"),
        "GET /department HTTP/1.1" => response_text("Computer Science and Engineering"),
        _ => response_text("404 NOT FOUND"),
    };

    stream.write_all(response.as_bytes()).unwrap();
}
```

The *handle_connection* function takes as input a *TcpStream*, extracts the lines representing the request from it in *request_lines* and uses the *match* construct of rust to check what is the request, and then returns the string corresponding to the request.

server > src > lib.rs > { } impl ThreadPool > execute

```rust
pub struct ThreadPool{
    workers:Vec<Worker>,
    sender : mpsc::Sender<Job>,
}

type Job = Box<dyn FnOnce() + Send + 'static>;

struct Worker{
    id : usize,
    thread: thread::JoinHandle<()>,
}

impl Worker{
    pub fn new(id:usize,receiver: Arc<Mutex<mpsc::Receiver<Job>>>)->Worker{
        let thread= thread::spawn(move||loop{
            let job=receiver.lock().unwrap().recv().unwrap();
            println!("Worker {id} got a job, executing!");
            job();
        });
        return Worker{id,thread};
    }
}
```

server > src > lib.rs > { } impl ThreadPool > execute

```rust

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        assert!(size>0);
        let mut workers=Vec::with_capacity(size);
        let (sender,receiver)=mpsc::channel();
        let receiver = Arc::new(Mutex::new(receiver));
        for id in 0..size{
            workers.push(Worker::new(id,Arc::clone(&receiver)));
        }
        ThreadPool{workers,sender}
    }
    pub fn execute<F> (&self, f:F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);
        self.sender.send(job).unwrap();
    }
}
```

The struct *ThreadPool* contains a vector of workers and the sender end of the channel which is used for communication. Each worker contains an id and a thread to execute the task.

We haven't defined an array of threads in the struct *ThreadPool* as a thread has to be given the closure it has to execute at the time of declaration but we are making a pool of 4 threads, so instead we define an intermediate *Worker* between the *ThreadPool* and the threads. Each worker has a thread and an id, so basically we create a pool of 4 threads (workers) and the sending end of the channel in the pool. Whenever a query comes, it is sent to the workers so the worker has the receiver end. To ensure that each query is sent to only one worker, *Arc<Mutex<T>>* is used.

Rust's Borrow Checker:

Rust's borrow checker enforces strict ownership and borrowing rules at **compile time**, preventing **dangling pointers, double frees, and data races**.

Key Concepts : Ownership, Borrowing and Lifetimes

Rust's borrow checker ensures memory safety as:

1. It ensures ownership of any memory (heap or stack) is in one place. Only the owner of the data can manipulate it. Multiple read-only access is allowed.
2. Rust keeps track of the memory variables used and automatically frees up memory when all the variables using that memory are out of scope — whether on the stack or the heap.
3. Most of this happens at compile time — we do not have a runtime garbage collector that has to run alongside us, and there are no interruptions to the program execution to allow a garbage collector to interrogate memory usage.

Safety advantages of the Borrow Checker :

- Prevents Use-After-Free Errors

examples.rs > ...

```rust
fn main() {
let s1 = String::from("hello");
let s2 = s1; // Ownership moved to s2
println!("{}", s1); // ERROR: Value moved, use after move
}

```

- Avoid Dangling Pointers

examples.rs > ...

```rust
fn main() {
    let r;
    {
        let x = 5;
        r = &x; // ERROR: `x` does not live long enough
    }
    println!("{}", r);
}

```

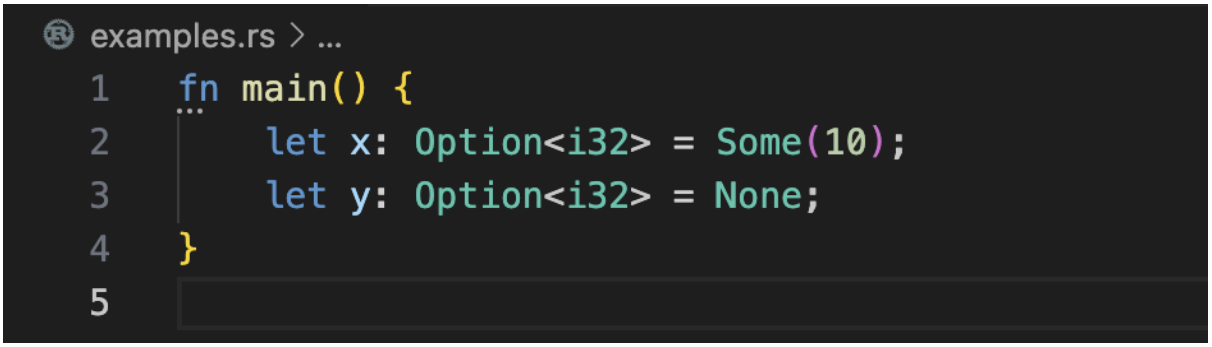
- Prevents Data Races

examples.rs > main

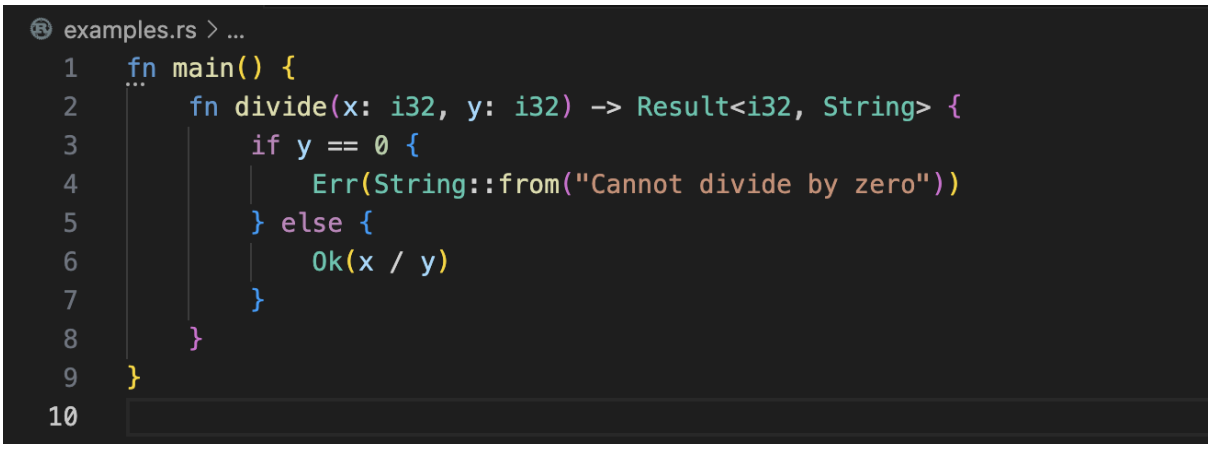
```rust
fn main() {
    let mut x = 10;
let r1 = &x;
let r2 = &x;
let r3 = &mut x; // ERROR: Cannot borrow mutably while borrowed immutably
}

```

Rust's Type Safety:
Rust's strong, static type system ensures that **type mismatches** and **invalid operations** are caught at compile time.

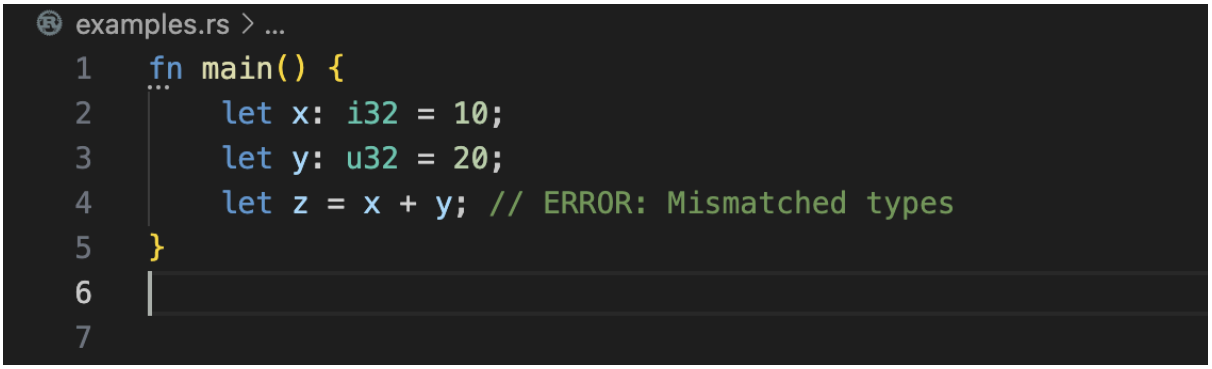
- No Null Pointers (Option<T>)



- Safe Pattern Matching for Error Handling (Result<T, E>)



- Strong Type Inference with Explicit Safety



- Preventing Buffer Overflows with Array Indexing Checks

Major issue faced while implementing : CORS