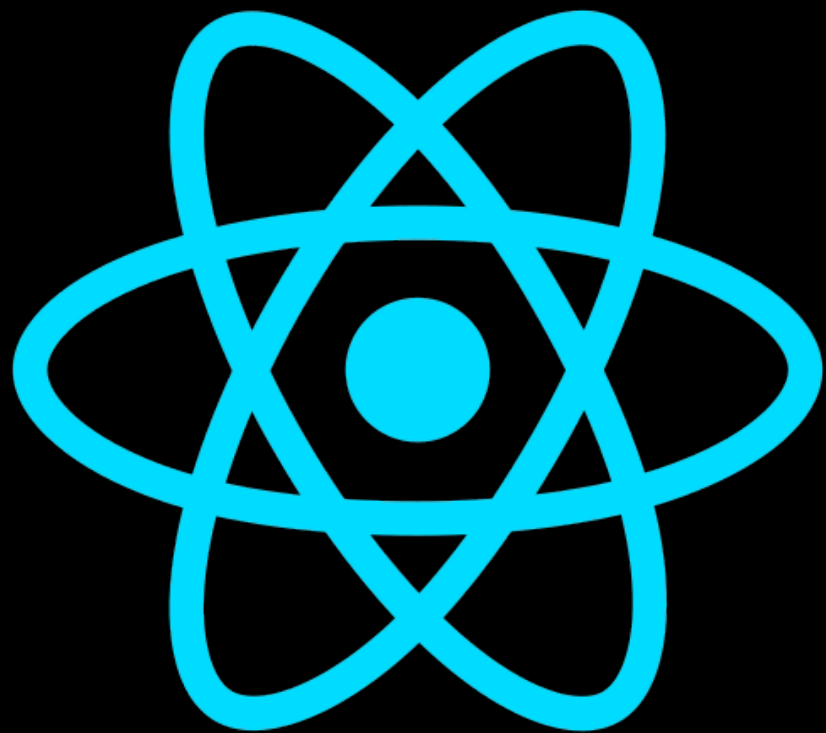


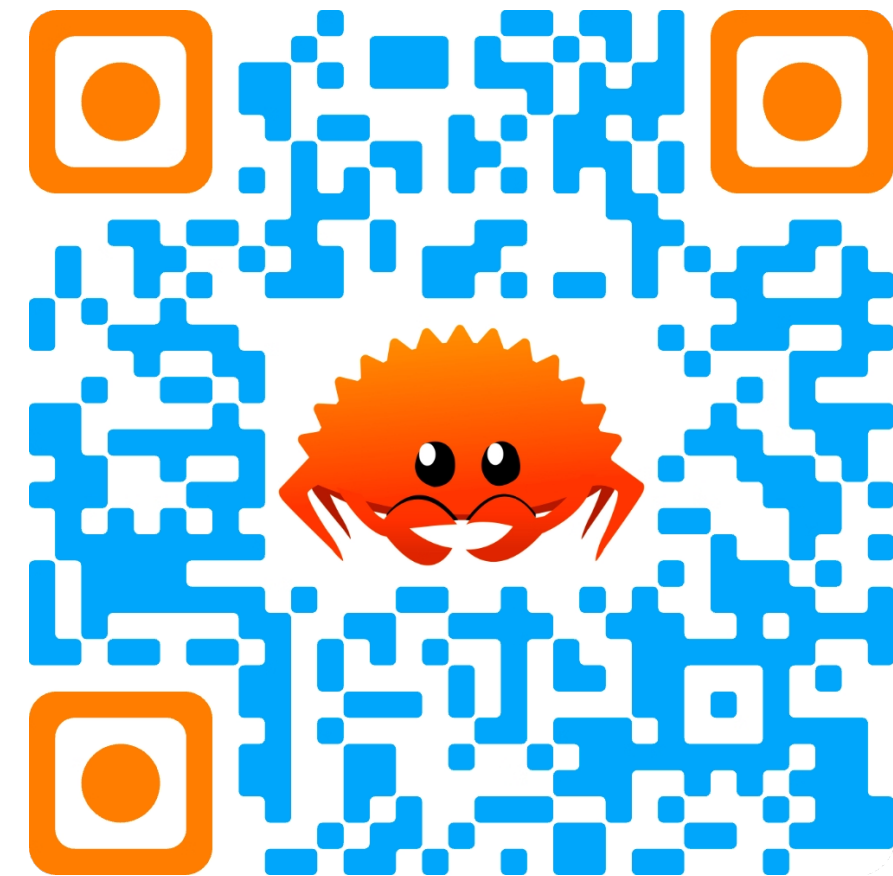
A SOY DEV'S FIRST FORAY INTO RUST

By Manik Rana



AGENDA

- Hi, I'm Manik
- Advent of Code (and failing)
- Building a TCP Echo Server (and failing)
- ~~Intro to Tauri~~
- ~~Making your own Tauri App~~



[slides and code](#)

HI, I'M MANIK

- Background:
 - Mostly a soy dev (React, Nextjs, Python)
 - I watch a lot of Primeagen
 - Dogs are awesome

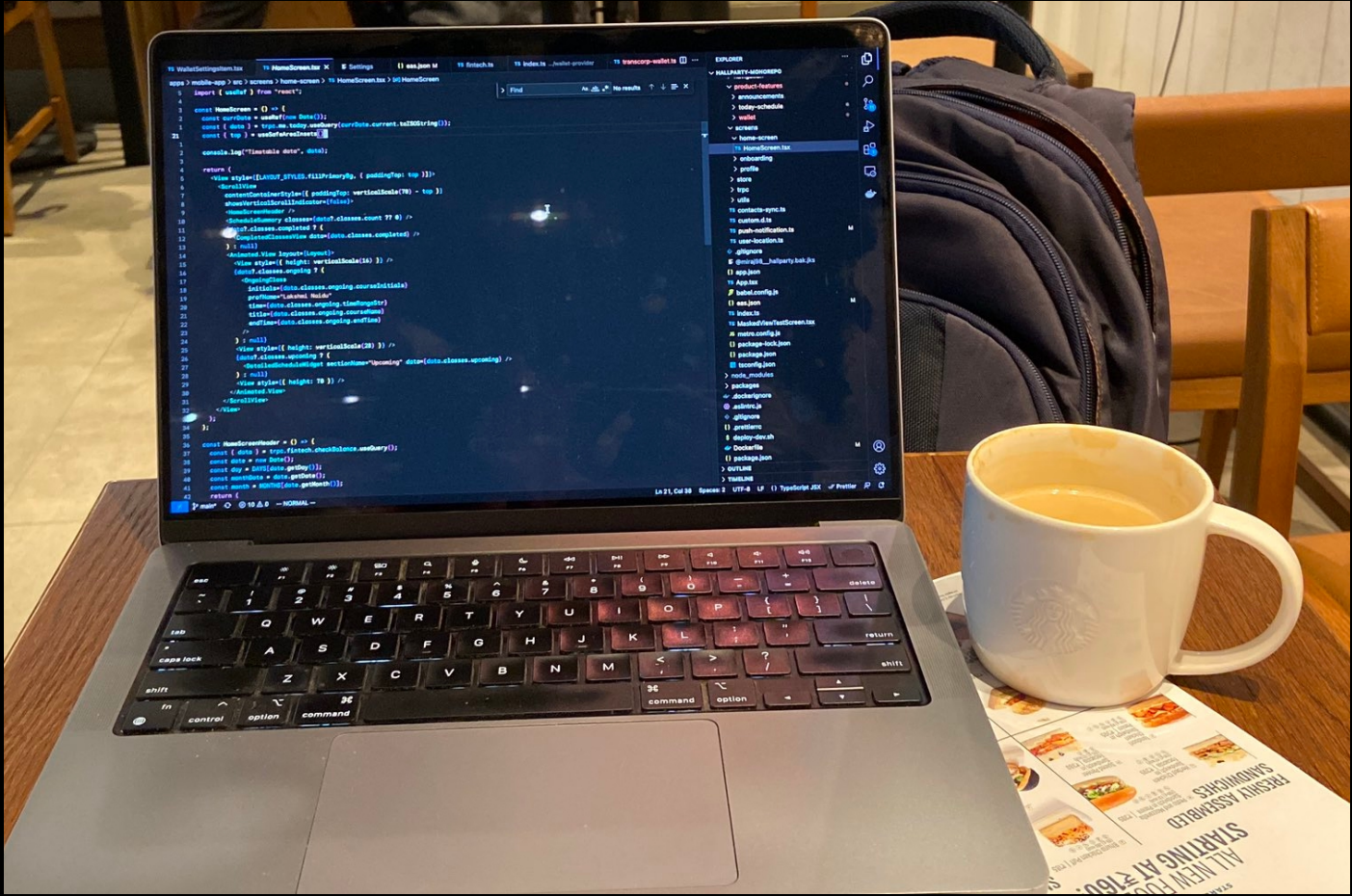


HI, I'M MANIK

- I'm here cause I've been banging my head against a wall trying to write rust and I don't want that to happen to you



SO WHY RUST



ME



WHAT IM TRYING TO BE

SO WHY RUST



A screenshot of a web browser's developer console. The console has tabs for 'Console', 'Search', and 'Emula'. Below the tabs, there are icons for a filter and a frame selector showing '<top frame>'. The console contains five lines of JavaScript code and their corresponding boolean outputs:

```
> 0 > null  
false  
> 0 >= null  
true  
> 0 == null  
false  
> 0 <= null  
true  
> 0 < null  
false
```

what???

SO WHY RUST

Imao

```
Console Search Emula
<top frame>
> 0 > null
false
> 0 >= null
true
> 0 == null
false
> 0 <= null
true
> 0 < null
false
```

what???

`x <= y` is generally equivalent to `x < y || x == y`, except for a few cases:

- When one of `x` or `y` is `null`, and the other is something that's not `null` and becomes 0 when [coerced to numeric](#) (including `0`, `0n`, `false`, `""`, `"0"`, `new Date(0)`, etc.): `x <= y` is `true`, while `x < y || x == y` is `false`.
- When one of `x` or `y` is `undefined`, and the other is one of `null` or `undefined`: `x <= y` is `false`, while `x == y` is `true`.
- When `x` and `y` are the same object that becomes `NaN` after the first step of [Less than](#) (such as `new Date(NaN)`): `x <= y` is `false`, while `x == y` is `true`.
- When `x` and `y` are different objects that become the same value after the first step of [Less than](#): `x <= y` is `true`, while `x < y || x == y` is `false`.

SO HOW RUST

- The Rust Book <https://doc.rust-lang.org/book/>
 - Everything you need to learn rust
- Rustlings <https://github.com/rust-lang/rustlings>
 - Learn rust by solving exercises
- MIT's Rust book: <https://shorturl.at/jrG57>

The Rust Programming Language

by Steve Klabnik and Carol Nichols, with contributions from the Rust Community

This version of the text assumes you're using Rust 1.67.1 (released 2023-02-09) or later. See the ["Installation" section of Chapter 1](#) to install or update Rust.

The HTML format is available online at <https://doc.rust-lang.org/stable/book/> and offline with installations of Rust made with `rustup ; run rustup docs --book` to open.

Several community [translations](#) are also available.

This text is available in [paperback](#) and [ebook](#) format from No Starch Press.

🔥 Want a more interactive learning experience? Try out a different version of the Rust Book, featuring: quizzes, highlighting, visualizations, and more: <https://rust-book.cs.brown.edu>

rust-lang/rustlings



🦀 Small exercises to get you used to reading and writing Rust code!

👤 376
Contributors

📦 7k
Used by

★ 45k
Stars

🔗 8k
Forks



SO WHEN RUST

- This month I tried solving Advent of Code
 - It was a disaster

Not too bad ngl



BASIC SETUP

main function

- Reads input txt
- calls solution function
- prints output

cargo run —bin part1.rs

solution

- magic goes here

test

- runs a unit test by calling
cargo test —bin part1.rs

```
part1.rs M x
2023 > day-01 > src > bin > part1.rs > ...
You, 1 second ago | 1 author (You)
1 fn main() {
2     let input = include_str!("./input1.txt");
3     let output = part1(input);
4     dbg!(output);
5 }
6
7 fn part1(input: &str) -> i32 {
8     let mut res = 0;
9     for line in input.lines() {
10         // convert lines to list of chars
11         let chars: Vec<char> = line.chars().collect();
12         // remove all non-numeric chars
13         let nums: Vec<char> = chars.into_iter().filter(|c| c.is_numeric()).collect();
14         // get first and last items and convert to i32
15         let first = nums.first().unwrap().to_digit(10).unwrap() as i32;
16         let last = nums.last().unwrap().to_digit(10).unwrap() as i32;
17         let num = first * 10 + last;
18
19         res += num;
20     }
21     res
22 }
23
24 You, 1 second ago | 1 author (You)
25 #[cfg(test)]
26 mod test {
27     use super::*;
28
29     #[test]
30     fn test1() {
31         let result = part1("1abc2
32 pqr3stu8vwx
33 a1b2c3d4e5f
34 treb7uchet");
35         assert_eq!(result, 142);
36     }
37 }
```

AOC DAY 1 RECAP

- Get the first and last number from each line of the input
- Convert it into a 2 digit number
- Add up all the numbers

1abc2

pqr3stu8vwx

a1b2c3d4e5f

treb7uchet

1abc2

pqr3stu8vwx

a1b2c3d4e5f

treb7uchet

12

38

15

77

142

BASIC SOLUTION

```
fn part1(input: &str) -> i32 {  
    let mut res = 0;  
    for line in input.lines() {  
        // convert lines to list of chars  
        let chars: Vec<char> = line.chars().collect();  
        // remove all non-numeric chars  
        let nums: Vec<char> = chars.into_iter().filter(|c| c.is_numeric()).collect();  
        // get first and last items and convert to i32  
        let first = nums.first().unwrap().to_digit(10).unwrap() as i32;  
        let last = nums.last().unwrap().to_digit(10).unwrap() as i32;  
        let num = first * 10 + last;  
  
        res += num;  
    }  
    res  
}
```

WHAT ARE THESE?

```
fn part1(input: &str) -> i32 {  
    let mut res = 0;  
    for line in input.lines() {  
        // convert lines to list of chars  
        let chars: Vec<char> = line.chars().collect();  
        // remove all non-numeric chars  
        let nums: Vec<char> = chars.into_iter().filter(|c| c.is_numeric()).collect();  
        // get first and last items and convert to i32  
        let first = nums.first().unwrap().to_digit(10).unwrap() as i32;  
        let last = nums.last().unwrap().to_digit(10).unwrap() as i32;  
        let num = first * 10 + last;  
  
        res += num;  
    }  
    res  
}
```

WHAT ARE THESE?

- `.into_iter()` lets you iterate over data, regardless of its type
- `.collect()` transforms an iterator into a collection, collecting all the items from the iterator. It's a general-purpose function that's used at the end of a chain of iterator functions.
- `.unwrap()` In Rust, "unwrapping" something means to say, "Give me the result of the computation, and if there was an error, panic and stop the program". – MIT rust book

.unwrap() and Option<T>

- Rust doesn't have nullable types.
- Rust uses the null-safe type Option<T>. Rust's pointer types must always point to a valid location; there are no "null" references.
- With Option, catch the failure instead of calling panic!

```
// get first and last items and convert to i32  
let first: Option<&char> = nums.first();
```

```
let first: &char = nums.first().unwrap();
```

```
core::option  
pub enum Option<T> {  
    None,  
    Some(T),  
}
```

- Unwrap either gets the value or panics



**WOW THAT WAS A
LOT OF RUST**

EVEN MORE RUST

- I also made a TCP echo server
- Requirements:
 - Implement TCP Echo Service from [RFC 862](#)
 - handle at least 5 simultaneous clients

RFC 862

TCP Based Echo Service

One echo service is defined as a connection based application on TCP. A server **listens for TCP connections** on TCP port 7. Once a connection is established **any data received is sent back**. This continues until the calling user terminates the connection.

```
#[tokio::main]
async fn main() -> io::Result<()> {
    // Bind the listener to the address
    let listener = TcpListener::bind("0.0.0.0:8080").await?;

    println!("Server listening on: {}", &listener.local_addr()?);

    loop {
        match listener.accept().await {
            Ok((mut socket, addr)) => {
                tokio::spawn(async move { echo(&mut socket, addr).await });
            }
            Err(e) => println!("Error accepting connection: {}", e),
        }
    }
}
```

Rust has Async!

```
#[tokio::main]
async fn main() -> io::Result<()> {
    // Bind the listener to the address
    let listener = TcpListener::bind("0.0.0.0:8080").await;

    println!("Server listening on: {}", &listener.local_addr()?);

    loop {
        match listener.accept().await {
            Ok((mut socket, addr)) => {
                tokio::spawn(async move { echo(&mut socket, addr).await });
            }
            Err(e) => println!("Error accepting connection: {}", e),
        }
    }
}
```

async allows tasks to run concurrently or in a non-blocking manner.

ASYNC AWAIT, FUTURES

In Rust, the Future trait represents a computation that may not have completed yet but will eventually produce a result or an error.

It defines two associated types:

- Output (the type of the eventual result)
- Poll (a method used to drive the computation towards completion).

`async fn` implicitly return futures

```
...  
pub trait Future {  
    type Output;  
  
    // Required method  
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;  
}
```

Always has been

Wait, its all



W3Schools

<https://www.w3schools.com> › js › js_promise

JavaScript Promises

?

VALID



```
async fn hello() -> String {  
    // Sleep for 2 seconds  
    sleep(Duration::from_secs(2)).await;  
  
    // Return "Hello world"  
    "Hello world".to_string()  
}
```

This is valid code that the compiler won't complain about

INVALID



```
async fn main() {  
    let result = hello().await;  
    println!("{}", result);  
}
```

Can't run async functions in main without a runtime

VALID



```
async fn hello() -> String {  
    // Sleep for 2 seconds  
    sleep(Duration::from_secs(2)).await;  
  
    // Return "Hello world"  
    "Hello world".to_string()  
}
```

VALID



```
use tokio::time::sleep;  
  
#[tokio::main]  
async fn main() {  
    let result = hello().await;  
    println!("{}", result);  
}
```

Rust now has a runtime for
async functions

TOKIO

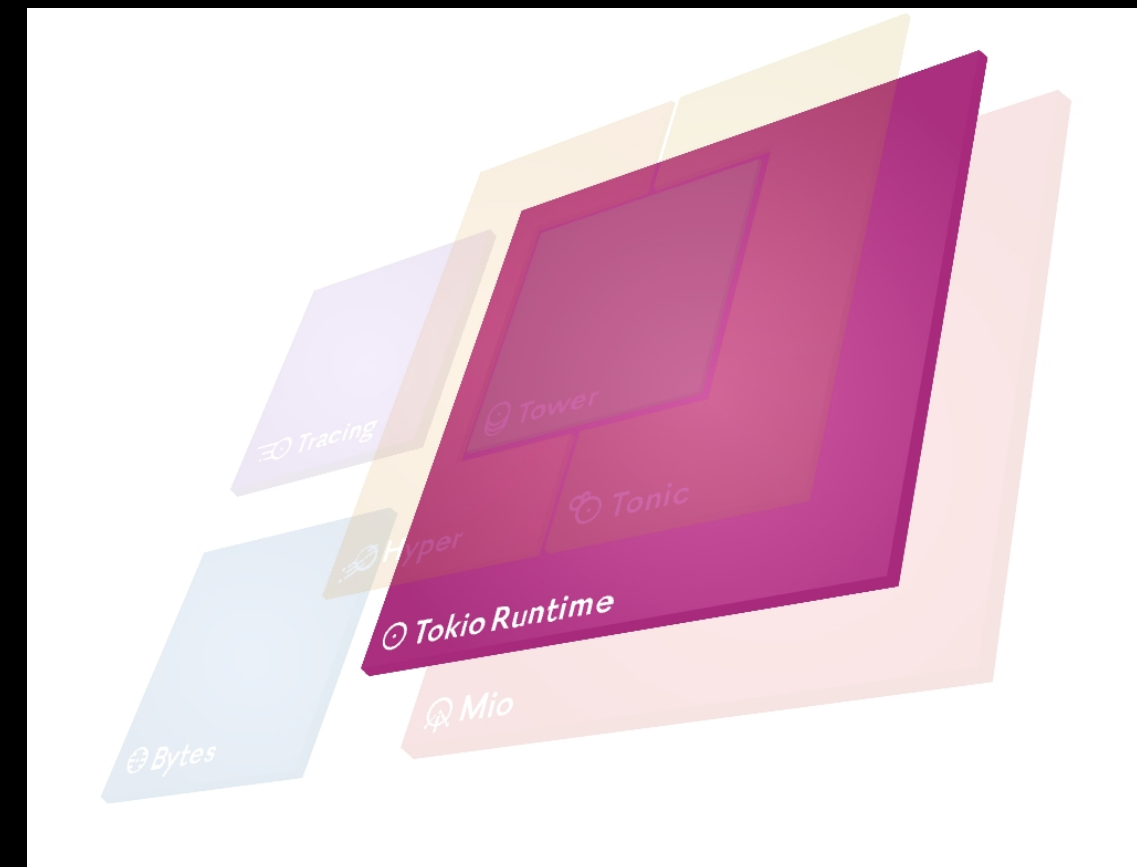
Tokio is an asynchronous runtime for Rust.

It provides the building blocks needed for writing network applications.

Rust itself has support for asynchronous programming through its `async` and `await` syntax, allowing you to write asynchronous code using the `async fn` and `.await` constructs.

However, **asynchronous I/O requires more than just language-level support; it also needs an asynchronous runtime** to handle scheduling, managing tasks, and handling I/O efficiently.

This is where tokio comes in.



OK, BACK TO THE ECHO SERVER

```
#[tokio::main]
async fn main() -> io::Result<()> {
    // Bind the listener to the address
    let listener = TcpListener::bind("0.0.0.0:8080").await?;

    println!("Server listening on: {}", &listener.local_addr()?);

    loop {
        match listener.accept().await {
            Ok((mut socket, addr)) => {
                tokio::spawn(async move { echo(&mut socket, addr).await });
            }
            Err(e) => println!("Error accepting connection: {}", e),
        }
    }
}
```

bind a TCP Listener to
localhost:8080 to handle
incoming connections

```
#[tokio::main]
async fn main() -> io::Result<()> {
    // Bind the listener to the address
    let listener = TcpListener::bind("0.0.0.0:8080").await?;

    println!("Server listening on: {}", &listener.local_addr()?);

    loop {
        match listener.accept().await {
            Ok((mut socket, addr)) => {
                tokio::spawn(async move { echo(&mut socket, addr).await });
            }
            Err(e) => println!("Error accepting connection: {}", e),
        }
    }
}
```

await incoming
connections in a
loop

call the echo function on
accepted socket

```
#[tokio::main]
async fn main() -> io::Result<()> {
    // Bind the listener to the address
    let listener = TcpListener::bind("0.0.0.0:8080").await?;

    println!("Server listening on: {}", &listener.local_addr()?);

    loop {
        match listener.accept().await {
            Ok((mut socket, addr)) => {
                tokio::spawn(async move { echo(&mut socket, addr).await });
            }
            Err(e) => println!("Error accepting connection: {}", e),
        }
    }
}
```



Gives the echo fn access to the outer lexical environment

```

async fn echo(socket: &mut TcpStream, address: SocketAddr) -> io::Result<()> {
    println!("New client: {}", address);

    let mut buffer = [0; 1024]; // Buffer to read/write data

    println!("Connected: {}", address);

    loop {
        let bytes_read = match socket.read(&mut buffer).await {
            Ok(n) if n == 0 => break, // End of stream
            Ok(n) => n,
            Err(e) => {
                eprintln!("Failed to read from socket: {}", e);
                return Err(e);
            }
        };

        match socket.write_all(&buffer[..bytes_read]).await {
            Ok(_) => {} // Data successfully written
            Err(e) => {
                eprintln!("Failed to write to socket: {}", e);
                return Err(e);
            }
        }
    }

    println!("Closed connection: {}", address);
    Ok(())
}

```

```

async fn echo(socket: &mut TcpStream, address: SocketAddr) -> io::Result<()> {
    println!("New client: {}", address);

    let mut buffer = [0; 1024]; // Buffer to read/write data

    println!("Connected: {}", address);

    loop {
        let bytes_read = match socket.read(&mut buffer).await {
            Ok(n) if n == 0 => break, // End of stream
            Ok(n) => n,
            Err(e) => {
                eprintln!("Failed to read from socket: {}", e);
                return Err(e);
            }
        };

        match socket.write_all(&buffer[..bytes_read]).await {
            Ok(_) => {} // Data successfully written
            Err(e) => {
                eprintln!("Failed to write to socket: {}", e);
                return Err(e);
            }
        }
    }

    println!("Closed connection: {}", address);
    Ok(())
}

```

A TCP Receive Window is a buffer that temporarily stores incoming data on each side of a TCP connection. TCP buffers data because it doesn't know when the application will read it

```

async fn echo(socket: &mut TcpStream, address: SocketAddr) -> io::Result<()> {
    println!("New client: {}", address);

    let mut buffer = [0; 1024]; // Buffer to read/write data

    println!("Connected: {}", address);

    loop {
        let bytes_read = match socket.read(&mut buffer).await {
            Ok(n) if n == 0 => break, // End of stream
            Ok(n) => n,
            Err(e) => {
                eprintln!("Failed to read from socket: {}", e);
                return Err(e);
            }
        };

        match socket.write_all(&buffer[..bytes_read]).await {
            Ok(_) => {} // Data successfully written
            Err(e) => {
                eprintln!("Failed to write to socket: {}", e);
                return Err(e);
            }
        }
    }

    println!("Closed connection: {}", address);
    Ok(())
}

```

A TCP Receive Window is a buffer that temporarily stores incoming data on each side of a TCP connection. TCP buffers data because it doesn't know when the application will read it

for each request, we will read the socket (TcpStream).

If we read nothing, we close the connection

```

async fn echo(socket: &mut TcpStream, address: SocketAddr) -> io::Result<()> {
    println!("New client: {}", address);

    let mut buffer = [0; 1024]; // Buffer to read/write data

    println!("Connected: {}", address);

    loop {
        let bytes_read = match socket.read(&mut buffer).await {
            Ok(n) if n == 0 => break, // End of stream
            Ok(n) => n,
            Err(e) => {
                eprintln!("Failed to read from socket: {}", e);
                return Err(e);
            }
        };

        match socket.write_all(&buffer[..bytes_read]).await {
            Ok(_) => {} // Data successfully written
            Err(e) => {
                eprintln!("Failed to write to socket: {}", e);
                return Err(e);
            }
        }
    }

    println!("Closed connection: {}", address);
    Ok(())
}

```

A TCP Receive Window is a buffer that temporarily stores incoming data on each side of a TCP connection. TCP buffers data because it doesn't know when the application will read it

for each request, we will read the socket (TcpStream)

If we read nothing, we close the connection

write all bytes in the buffer that have been read back to the client socket

close the connection

FULL SERVER CODE

```
use tokio::net::{TcpListener, TcpStream};
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use std::net::SocketAddr;
use std::io;

#[tokio::main]
▶ Run | Debug
async fn main() -> io::Result<()> {
    // Bind the listener to the address
    let listener: TcpListener = TcpListener::bind("0.0.0.0:8080").await?;

    println!("Server listening on: {}", &listener.local_addr()?);

    loop {
        match listener.accept().await {
            Ok((mut socket: TcpStream, addr: SocketAddr)) => {
                tokio::spawn(future: async move { echo(&mut socket, address: addr).await });
            }
            Err(e: Error) => println!("Error accepting connection: {}", e),
        }
    }
}
fn main
```

```
async fn echo(socket: &mut TcpStream, address: SocketAddr) -> io::Result<()> {
    println!("New client: {}", address);

    let mut buffer: [u8; 1024] = [0; 1024]; // Buffer to read/write data

    println!("Connected: {}", address);

    loop {
        let bytes_read: usize = match socket.read(buf: &mut buffer).await {
            Ok(n: usize) if n == 0 => break, // End of stream
            Ok(n: usize) => n,
            Err(e: Error) => {
                eprintln!("Failed to read from socket: {}", e);
                return Err(e);
            }
        };

        match socket.write_all(src: &buffer[..bytes_read]).await {
            Ok(_) => {} // Data successfully written
            Err(e: Error) => {
                eprintln!("Failed to write to socket: {}", e);
                return Err(e);
            }
        }
    }

    println!("Closed connection: {}", address);
    Ok(())
}
fn echo
```

**OK, WOW
THAT WAS A LOT**

LETS CHILL

FUN FACT: I'M GETTING BRACES TMR

SO WHAT DID WE COVER?

- How I setup rust for solving AoC.
- Dissecting AoC day_1
- .unwrap(), .collect(), .into_iter(), and other weird stuff like no nulls in Rust
- Async/Await in Rust with Tokio
- Building a TCP echo service the rust book also covers this
liked TCP servers? Try Prothackers

AGENDA

BUT WHERE DID I FAIL?

- Hi, I'm Manik
- Advent of Code (and failing)
- Building a TCP Echo Server (and failing)
- ~~Intro to Tauri~~
- ~~Making your own Tauri App~~

FIN