

Forum:[Hack Forums](#)

Thread: [\[Rust\] Create an advanced Logger in Rust - Part 1](#)

Infinity. - March 19, 2025, 5:49 am

Hey There, In this guide, I'll simplify log management for any application you want to develop. This will be Part 1 of the series, where I'll cover the fundamentals and best practices.

Here's what I have in mind: Create an advance logger Part 1- This thread
Create an advance logger Part 2 - Work more on asynchronous Logging
Create an advance logger Part 2- Extend our Logger to a KeyLogger Project

I will roughly cover the following features in Part 1. In Part 2 we shall work on advancing what we cover today.

Colored Terminal Output:

- Log levels are colored in the terminal output (e.g., errors in red, warnings in yellow).

Rotating Log Files:

- Logs are rotated when they reach 1 MB in size, and up to 5 old log files are kept.
- Logs are stored in the logs/ directory.

Custom Filters:

-Only logs from specific modules (my_module and another_module) are allowed. (this is yet to be done)

Asynchronous Logging:

- The tracing crate is used for asynchronous logging.
- Logs are sent to a remote server asynchronously using request.

Multiple Destinations:

- Send logs to file, terminal or remote services

How are our finished app will give us the errors we need

1. Terminal Logs

2. Logs sent to remote server

3. File Logs

Let's Jump to it!

Required Dependencies

To start, lets add the following dependencies to your Cargo.toml file:

```

Code [dependencies]
log = "0.4"
fern = "0.6"
log4rs = "1.0"
chrono = "0.4"
tracing = "0.1"
tracing-subscriber = "0.3"
request = { version = "0.11", features = ["json"] }
tokio = { version = "1.0", features = ["full"] }
colored = "2.0"
[Setting Up a Terminal Logger

```

in our `terminal_logger` function we are configuring logging to display colored messages in the terminal:

```

Code fn terminal_logger() -> Result<(), fern::InitError> {
    Dispatch::new()
        .format(|out, message, record| {
            let level_color = match record.level() {
                log::Level::Error => "red",
                log::Level::Warn => "yellow",
                log::Level::Info => "green",
                log::Level::Debug => "blue",
                log::Level::Trace => "magenta",
            };
            out.finish(format_args!(
                "{} [{}] [{}]",
                chrono::Local::now().format("%Y-%m-%d %H:%M:%S"),
                record.level().to_string().color(level_color),
                record.target(),
                message
            ))
        })
        .chain(std::io::stdout()) // Log to stdout with colors
        .apply()?;
    Ok(())
}

```

This function: Formats log messages with timestamps and colors.
 Uses fern to manage log levels.
 Sends log messages to the terminal.

Configuring a Rotating Logger

A rotating logger saves logs to a file and rotates them when they reach a certain size.

```

Code fn rotating_logger() -> Result<(), Box<dyn std::error::Error>> {
    let window_roller = FixedWindowRoller::builder()
        .build("logs/app.log", 5)?;
    let size_trigger = SizeTrigger::new(1024 * 1024); // Rotate after 1 MB
    let policy = CompoundPolicy::new(Box::new(size_trigger), Box::new(window_roller));

    let rolling_file = RollingFileAppender::builder()
        .encoder(Box::new(PatternEncoder::new("{d} [{l}] {m}{n}")))
        .build("logs/app.log", Box::new(policy))?;
}

```

```

let config = Config::builder()
    .appender(Appender::builder().build("rolling_file", Box::new(rolling_file)))
    .build(Root::builder().appender("rolling_file").build(LevelFilter::Info));

log4rs::init_config(config)?;
Ok()
}

```

Here in this function we are doing

Sets up a log file (logs/app.log).

Uses log4rs to create a rotating policy when the log file exceeds 1MB.

Saves up to 5 rotated log files.

Sending Logs to a Remote Server

Here we can send log messages to an external server asynchronously:

```

Code async fn send_log_to_server(level: &str, message: &str) {
    let client = Client::new();
    let _ = client.post("http://localhost:3000/logs")
        .json(&serde_json::json!({
            "level": level,
            "message": message,
        }))
        .send()
        .await;
}

```

Now let us initialize the type of logger we need

Initializing the Logger

```

Code fn init_logger() -> Result<(), Box> {
    rotating_logger()?;
    // terminal_logger()?;
    Ok()
}

```

This function initializes the logging system using the rotating file logger.

Putting it all together

Finally, we integrate logging into our main function:

```

Code fn main() {
    // Initialize the logger
    if let Err(e) = init_logger() {
        eprintln!("Failed to initialize logger: {}", e);
        return;
    }
    let a: u8 = 5;
    info!("LOOK! I logged an info message");
    info!("LOOK! I logged an info message again {}", a);
    error!("LOOK! NO! an error message");
    warn!("LOOK! A warning message");
    error!("LOOK! Yet another error!");
}

```

```
debug!("LOOK! I logged a debug message but it won't appear in the file due to log level");
```

```
// Asynchronous log example
let rt = Runtime::new().unwrap();
rt.block_on(async {
    trace_info!("This is an async info message using tracing crate");
    trace_error!("This is an async error message using tracing crate");

    send_log_to_server("info", "This log is sent to a remote server").await;
});
}
```

Here we call our initialize logger function and then we log messages at different levels (info, error, warn, debug). And a small demonstration of asynchronous logging with tracing and sending logs to a remote server.

I hope this was an explanation on my end, If I have missed out some important things required to explain, please post it in the thread and we shall resolve it

Infinity. - April 9, 2025, 11:27 am

So I literally get 0 response on this thread? huh? HF?

Stanley - April 9, 2025, 11:27 am

I'm Stanley HF's helper bot. The OP (original poster) of this thread wants everyone to see this page and has elected to bump it to the top of the showthread listing. Since HF doesn't want a ton of LQP (Low Quality Posts) this is the best method of bumping your content. There is a small ?yte fee for each bump. Any OP can bump any of their threads on HF from the linked icon on the top of their first post.

If you haven't already read the first post of this thread we encourage you to do so.

Thanks,
Stanley

Shrdw - April 9, 2025, 7:05 pm

(Mar 19, 2025 05:49 AM)Infinity. Wrote: [\[CITE\]](#)

Hey There, In this guide, I'll simplify log management for any application you want to develop. This will be Part 1 of the series, where I'll cover the fundamentals and best practices.

Here's what I have in mind: Create an advance logger Part 1- This thread
Create an advance logger Part 2 - Work more on asynchronous Logging
Create an advance logger Part 2- Extend our Logger to a KeyLogger Project

I will roughly cover the following features in Part 1. In Part 2 we shall work on advancing what we cover today.

Colored Terminal Output:

- Log levels are colored in the terminal output (e.g., errors in red, warnings in yellow).

Rotating Log Files:

- Logs are rotated when they reach 1 MB in size, and up to 5 old log files are kept.
- Logs are stored in the logs/ directory.

Custom Filters:

-Only logs from specific modules (my_module and another_module) are allowed. (this is yet to be done)

Asynchronous Logging:

- The tracing crate is used for asynchronous logging.
- Logs are sent to a remote server asynchronously using reqwest.

Multiple Destinations:

- Send logs to file, terminal or remote services

How are our finished app will give us the errors we need

1. Terminal Logs

2. Logs sent to remote server

3. File Logs

Let's Jump to it!

Required Dependencies

To start, lets add the following dependencies to your Cargo.toml file:

Code [dependencies]

```
log = "0.4"
fern = "0.6"
log4rs = "1.0"
chrono = "0.4"
tracing = "0.1"
tracing-subscriber = "0.3"
reqwest = { version = "0.11", features = ["json"] }
tokio = { version = "1.0", features = ["full"] }
colored = "2.0"
```

[Setting Up a Terminal Logger

in our terminal_logger function we are configuring logging to display colored messages in the terminal:

```
Code fn terminal_logger() -> Result<(), fern::InitError> {
    Dispatch::new()
        .format(|out, message, record| {
            let level_color = match record.level() {
                log::Level::Error => "red",
                log::Level::Warn => "yellow",
                log::Level::Info => "green",
                log::Level::Debug => "blue",
```

```

        log::Level::Trace => "magenta",
    };
    out.finish(format_args!(
        "{} [{}] [{}] {}",
        chrono::Local::now().format("%Y-%m-%d %H:%M:%S"),
        record.level().to_string().color(level_color),
        record.target(),
        message
    ))
}
.chain(std::io::stdout()) // Log to stdout with colors
.apply()?;
Ok()
}

```

This function: Formats log messages with timestamps and colors.
 Uses fern to manage log levels.
 Sends log messages to the terminal.

Configuring a Rotating Logger

A rotating logger saves logs to a file and rotates them when they reach a certain size.

```

Code fn rotating_logger() -> Result<(), Box<dyn std::error::Error>> {
    let window_roller = FixedWindowRoller::builder()
        .build("logs/app.log.{}", 5)?;
    let size_trigger = SizeTrigger::new(1024 * 1024); // Rotate after 1 MB
    let policy = CompoundPolicy::new(Box::new(size_trigger), Box::new(window_roller));

    let rolling_file = RollingFileAppender::builder()
        .encoder(Box::new(PatternEncoder::new("{d} [{l}] {m}{n}")))
        .build("logs/app.log", Box::new(policy))?;

    let config = Config::builder()
        .appender(Appender::builder().build("rolling_file", Box::new(rolling_file)))
        .build(Root::builder().appender("rolling_file").build(LevelFilter::Info))?;

    log4rs::init_config(config)?;
    Ok()
}

```

Here in this function we are doing
 Sets up a log file (logs/app.log).
 Uses log4rs to create a rotating policy when the log file exceeds 1MB.
 Saves up to 5 rotated log files.

Sending Logs to a Remote Server

Here we can send log messages to an external server asynchronously:

```

Code async fn send_log_to_server(level: &str, message: &str) {
    let client = Client::new();
    let _ = client.post("http://localhost:3000/logs")
        .json(&serde_json::json!({

```

```

        "level": level,
        "message": message,
    )))
    .send()
    .await;
}

```

Now let us initialize the type of logger we need

Initializing the Logger

```

Code fn init_logger() -> Result<(), Box> {
    rotating_logger()?;
    // terminal_logger()?;
    Ok(())
}

```

This function initializes the logging system using the rotating file logger.

Putting it all together

Finally, we integrate logging into our main function:

```

Code fn main() {
    // Initialize the logger
    if let Err(e) = init_logger() {
        eprintln!("Failed to initialize logger: {}", e);
        return;
    }
    let a: u8 = 5;
    info!("LOOK! I logged an info message");
    info!("LOOK! I logged an info message again {}", a);
    error!("LOOK! NO! an error message");
    warn!("LOOK! A warning message");
    error!("LOOK! Yet another error!");
    debug!("LOOK! I logged a debug message but it won't appear in the file due to log level");

    // Asynchronous log example
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        trace_info!("This is an async info message using tracing crate");
        trace_error!("This is an async error message using tracing crate");

        send_log_to_server("info", "This log is sent to a remote server").await;
    });
}

```

Here we call our initialize logger function and then we log messages at different levels (info, error, warn, debug). And a small demonstration of asynchronous logging with tracing and sending logs to a remote server.

I hope this was an explanation on my end, If I have missed out some important things required to explain, please post it in the thread and we shall resolve it

I will not claim to be an expert programmer though my main question is:

Outside of creating the in initial logger in rust what are the pros and cons vs other languages as i understand a memory safe language is a plus in this case. Say over C++ or go lang?

Been working on an irc bot controller in golang and C++ Loader so i understand enough though my code is far from worth sharing.

I only mention this because i know very little about rust and this seems like an interesting project overall. Mainly how the keylogger would work better vs say c++ or other languages?

Klonopin - April 12, 2025, 6:59 am

(Apr 9, 2025 11:27 AM)Infinity. Wrote: [\[CITE\]](#) So I literally get 0 response on this thread? huh? HF? Yep, don't be surprised. development on HF is almost completely dead.

But nice post man, I originally thought this was gonna be a key logger. but still very nice!

A cool improvement would be implementing a `Caller Location Information` automatically capture and display the source file and line number from where a log message was issued.

example: ``[DEBUG] - src/main.rs:34 - starting the application``

(Apr 9, 2025 07:05 PM)Shrdw Wrote: [\[CITE\]](#) Mainly how the keylogger would work better vs say c++ or other languages?

This isn't a key logger.

Anonymuk - September 24, 2026, 10:01 am

I must start looking into this RUST. I see a lot of people using it.... including Linux Devs