Ca' Foscari
University
of Venice

Master's Degree programme

Computer Science and Information Technology - CM90

CronFrame: A Macro Annotation Cron Job Framework with Web Server and CLI Tool written in Rust

Supervisor
Ch. Prof. Pietro Ferrara

Assistant supervisor
Ch. Prof. Gianluca Caiazza

Graduand
Antonio Cimino
Matriculation Number 897613

Academic Year
2023 / 2024

Contents

Appendix Contents

1. Introduction

This thesis focuses on presenting and describing CronFrame, a framework and CLI tool for the definition and execution of cron jobs developed in the Rust language to be used in Rust projects.

Cron jobs were created to solve the problem of having any kind of task executed at specific times with a simple syntax for the job definition. The first program that addressed this need was aptly named cron, from "Chronos" the Greek word for time, and made it possible to have jobs running with just a string that took care of its scheduling followed by the command to execute. Cron and most other tools of this kind address the execution of commands at the user level, which means a program is run as if it were entered in the terminal by the user. Therefore any command the user has got permission to run can be run. In the context of programming languages, this can be highly restrictive.

If we are dealing with interpreted languages then there is not much of a problem, the command is the call to the interpreter to run the script but if we look at compiled languages we can run their artifact, most of the time an executable binary, only after compilation has completed successfully.

This means that to run a job that executes said binary we require a first job to compile the source code, and a second job to run it.

However, doing so raises other problems:
- How can we be sure that compilation has terminated before it is scheduled to be run by cron?
- How can we check that there hasn't been a compilation error?

We can't if we just use these two commands and need workarounds.

One of these workarounds could be turning the cron job that runs the binary into a job that runs a script that checks compilation is done and only in that case runs the job. Or maybe get rid of the two jobs and have a single job running a script that checks whether compilation has already been performed and if so, run the job. But still, we are relying on a script that is possibly written in another language and interpreted.

This introduces an unnecessary "middleman" and complicates things more than they need to be. It is also tedious for anyone who wants to run jobs since it would require them to set up such scripts in the first place. The main reason behind CronFrame is to render all this nearly identical to simply running a binary in the original cron tool. Such problems in the framework are dealt with at compile time of the job so that it always has everything it needs to run at its disposal when the time comes, and if we need new jobs when the framework is already running we can always add them either from code or with the CLI tool.

CronFrame is primarily a framework for defining and executing cron jobs. It uses macro annotations as the main method of job definition but also supports job definition through function calls. This framework can be easily integrated into any Rust project, allowing a cron job scheduler to work with minimal configuration and setup. Upon initialization of the framework instance, a web server is launched, providing users with GUI functionalities and API routes.

The second aspect of Cronframe is that it serves as a CLI tool that uses the homonymous framework. This means that users can initiate the scheduler using just one command and utilize additional commands for job definition, such as loading them from a txt file containing a job list or adding them one by one in the terminal. The tool launches a global instance of CronFrame, which can be configured to some extent, offering all the functionalities of the framework along with some extra commands for added convenience.

The motivation behind this project stems from wanting to turn Rust code into a cron job by simply adding an annotation to the original code. This idea is inspired by frameworks like Spring, which utilize annotations to expand the capabilities of the host language, in the case of Spring, Java. Rust conveniently offers a native way to extend itself using macros, and certain types of macros can be employed like annotations in Spring.

CronFrame first started conceptually as a library but soon evolved into a small and compact framework. It can be easily extended and maintained by anyone with a decent familiarity with the Rust language. The CLI tool was a natural addition to the original framework.

Documentation of the framework is provided with the cargo doc tool native to Rust which allows for embedding documentation as comments on top of code.

A website containing step-by-step guided tutorials has been built with docusaurus and configured to provide references to multiple versions of the framework and CLI tool.

The entire code is available on a GitHub repository under either an MIT license or an APACHE 2.0 license.

Chapter 2 provides a general definition of a cronjob and then specializes it to the framework. Chapter 3 quickly delves into Rust and focuses on its macros to convey the reason behind its choice for the project. Chapter 4 presents the framework in all of its components explaining how they work and relate to each other. Chapter 5 is where the CLI tool is presented with a detailed explanation of its commands. Step-by-step guided tutorials for both the framework and the CLI tool are available in Chapter 6. The testing suite written for checking the correct functionality of the framework is described in Chapter 7. The reception both the framework and the CLI tool have received from the Rust community is recorded in Chapter 8. Finally, a development summary with notes and possible future outlooks is available in Chapter 9.

2. Scheduling of Cron Jobs

In this chapter, we start from the definition of a cron job. While at first, the definition will certainly be general, it will soon be specialized to the specific case of the project. Particular attention is given to the context of execution the framework operates in since it is not the typical one that might be anticipated by users of other cron job utilities.

2.1. Definition of a Cron Job

A cron job is a task of any type scheduled to run without ambiguity at a specific time. Such time of execution is described by a codified expression which allows the user to define schedules. A schedule can have times spanning from a simple one-off occurrence to multiple occurrences, to very frequent time intervals.

There are fields at the user's disposal for schedule definition such as minute, hour, month, and more. The exact number of fields is heavily dependent on the implementation of a specific tool. The values these fields can assume mostly follow those of the first release of the cron [1] CLI utility for Unix systems from 1975, but each cron job utility might differ in what is allowed in an expression, and even for the number of fields the expression is made of. One example is the mentioned cron tool lacking the field for seconds while more recent tools account for it. This is due to the use of different parsers and different needs at the time of writing a cron job tool.

For the entirety of this thesis, the composition of a cron expression will pertain to the specifics of the parser used which is the one implemented in the cron crate available on crates.io.

There are seven different fields at the user's disposal for schedule definition, these are:
- seconds
- minutes
- hours
- day of the month
- month
- day of the week
- year

2.1.1. The Schedule: Cron Expression

A string with at least six and at most seven fields separated by whitespace is known as a cron expression and it is the first step for the definition of a cron job since it takes care of denotating its schedule.

The fields are written from left to right with six of them as mandatory, the exception being the year field which can be omitted to mean any year.

All fields take numeric values in specific ranges. Only two fields, namely day of month and day of week can substitute their numeric values with a 3 letter abbreviation of their English names. These abbreviations are not case-sensitive.

All fields can have special characters that allow for more variation and flexibility when defining a schedule.

Field	Required	Allowed Values	Allowed Special Characters
Seconds	Yes	0-59	, - * /
Minutes	Yes	0-59	, - * /
Hours	Yes	0-23	, - * /
Day of Month	Yes	1-31	, - * / ?
Month	Yes	1-12 or JAN-DEC	, - * /
Day of Week	Yes	1-7 or sun-sat	, - * / ?
Year	No	empty or 1970-2100	, - * /

Table 1: Allowed Values for Cron Expressions Fields

The meaning of the special characters is quite straightforward and intuitive.

Character	Meaning	Effect
,	set of values	Only values separated by a comma are used
-	range of values	Only values in the range (inclusive) are used
*	wildcard / all values	All values the field can assume are considered valid
/	step of values	Start from the values on the left of the / and repeat after the time of the value on the right has elapsed
?	Yes	Does the same as * but can only be used in day of the month and day of the week

Table 2: Special Characters Meaning in Cron Expressions

Examples of job definitions with their worded-out description follow.

Cron Expression	Description
0/5 * * * * * *	At every 5th second from 0 through 59.
* 0 0 25 Dec * *	At 00:00 on day-of-month 25 in December.
* 0 22 * * mon-fri *	At 22:00:00 on every day-of-week from Monday through Friday.
* 30 5 * * SUN *	At 05:30 on Sunday.
* 0/10 7-9,16-18 * * Mon-Fri *	At every 10th minute from 0 through 59 past every hour from 7 through 9 and every hour from 16 through 18 on every day-of-week from Monday through Friday.
* 0 0,12 1,5,12 2 * *	At minute 0 past hour 0 and 12 on day-of-month 1, 5, and 12 in February.

Table 3: Examples of Cron Expressions

There are cron job tools that might provide further special characters to add more options. It is wise to never assume that every cron job tool provides the same functionality despite sharing a very similar base.

Some cron expressions define schedules that are of naturally frequent use, so much so that shorthands are available in their stead.

Cron Expression	Shorthand	Description
0 0 0 1 1 * *	@yearly	At midnight every January 1st
0 0 0 1 * * *	@monthly	At midnight every 1st day of every month
0 0 0 * * 1 *	@weekly	At midnight every week on Sunday
0 0 0 * * * *	@daily	At midnight every day
0 0 * * * * *	@hourly	At the start of every hour

Table 4: Shorthand of Common Cron Expressions

For shorthands as well, it varies on the tool implementation, some might have additional ones or even more than one alias for the same one.

2.1.2. The Job: Commands and Functions

With a cron expression at hand and our timezone of preference, for that will always be UTC, we can define schedules that best suit our needs. This however does not define a cron job in itself. In addition to the schedule a cron job needs, as the name implies, the job component to be executed. This job component is usually a CLI command in most cases which allows the user to define any possible task the computer has the software required to carry out, whether it be simply echoing some text or running a script to gather data or perform complex computations.

In the scope of this thesis however, a cron job is quite literally a function in the environment of the Rust programming language therefore what we can do in a job is dictated by the code the language enables us to write and what permissions we possess on the system.

2.2. Common Issues of Cron Jobs Usage

While the use of cron jobs is extremely common and has been trialed and tested for decades, there are still some pitfalls that users might find themself into when dealing with them:

- **Expression Issues**
  - Entering the wrong scheduling syntax, leading to jobs not running when intended or even entering invalid syntax, leading to a parse error.
- **Permissions Issues**
  - A cron job may not have the necessary access to execute if it is set up with the wrong user permissions.
- **Environment and Path Issues**
  - A cron job might fail due to it not having access to necessary environment variables.
  - It might also fail due to the use of relative paths instead of absolute paths in its code.
- **Overlapping Issues**
  - Without proper management, jobs may overlap, leading to system resource contention.
  - Even worse if jobs are performed in a cloud environment.
- **Error Handling Issues**
  - Without proper logging of output and errors it is troublesome to do any kind of troubleshooting.
  - Due to the jobs spanning different threads any troubleshooting is even more complex.
- **Output Issues**

  - If a cron job sends its output via email to one or multiple addresses it can be an issue if not properly configured, especially if the output volume is high.
- **Recovery Issues**
  - If a cron job fails to execute or complete either because of application or machine failure there is no guarantee it will be recovered in any way.
  - Job recovery is not standard functionality for most cron utilities, if required it must be handled by integrating a separate tool that takes care of it.

Dealing with all of these issues requires, for the most part, the application of best practices and common sense. Knowing the tools one uses in and out is usually the first step to take for mitigating or getting entirely rid of any kind of issue.

2.3. Execution of a Cron Job

Taking as example cron, one of the most used tools for cron jobs on Linux here is how to define a job using it:

```
* * * * * python3 /home/user/run_every_second.py
```

This definition is stored in what is called a crontab file and it will result in the execution of the Python script "run_every_second.py" every second by the cron utility that is automatically running right after the system starts.

Notice how the cron expression here is composed of 5 fields, in order from left to right:
- minute
- hour
- day-of-month
- month
- day-of-week

As stated earlier, cron doesn't account for the seconds field, and the year field is just omitted so the actual number of fields at a user's disposal when using cron is 6.

Our job here can be any command we have permission to run on the system:

```
* * * * * <command>
```

Since there is a plethora of time-withstanding tools that deal with this kind of cron job, this thesis work veered on making a framework for turning functions defined in the Rust language into cron jobs.

A closer depiction of how we define a cron job in this context would be:

```
* * * * * * * <function_name>
```

Notice how the fields in our case are 7 (if the year is not omitted).

This is however not the way to define a job because for us it needs context, namely the scope in which it can be defined and what it can access when dealing with struct types. Still, we will see that when using the CLI tool the job definition gets pretty similar to the one above and we can even use a txt file to define a list of jobs.

The CronFrame framework will make extensive use of macro annotations on top of functions for most of its functionality as will be explained in detail in Chapter 4 but before that a look as to why Rust was picked as a language for this project is dealt with in the following chapter.

3. The Rust Language

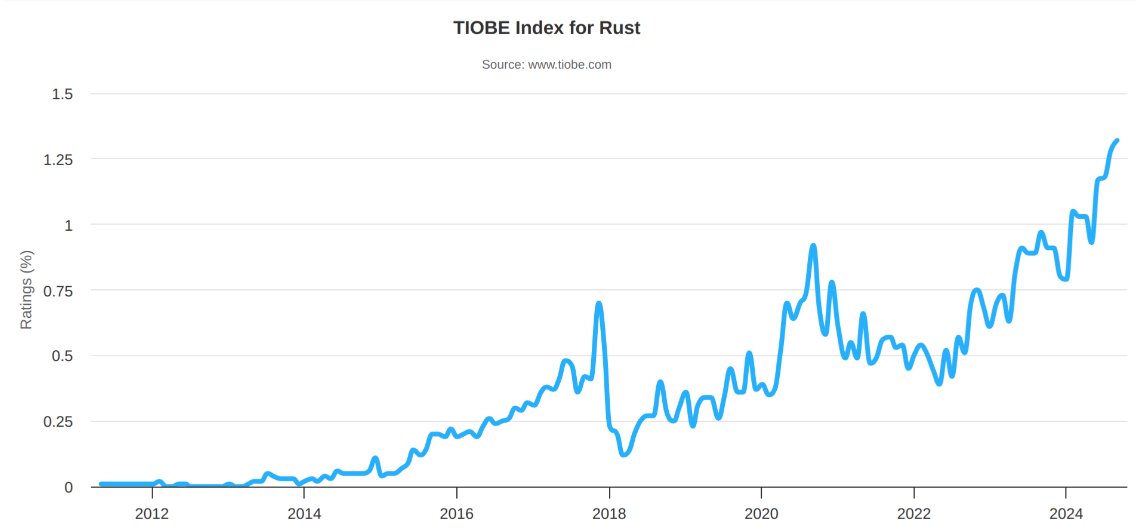
Before we start looking at cron jobs and the framework, this chapter introduces Rust to those who might be unfamiliar with it. Since Rust is a big and complex language, after a brief introduction focus is shifted to macros which play a big role in the framework.

Rust [2] is a general-purpose programming language that focuses on performance, concurrency, and memory safety. Memory safety in particular is enforced at compile time so that all references point to valid memory which is done without using a garbage collector.

All this is possible because of the borrow checker which tracks the object lifetime of all references during compilation helping prevent data races as well. Rust is heavily influenced by functional programming languages and has features such as immutability, higher-order functions, and algebraic data types. The programming paradigm of Rust is neither fully object-oriented nor fully functional but it can be used so to lean towards one of the two according to the user's preference.

It also comes with great tooling and dealing with structurally complex Rust projects is much less of a headache thanks to cargo [3], its native package manager whose counterpart in C and C++ is entirely lacking. Documentation also gets far less tedious to write since it can easily be written with doc comments on top of code with one neat feature of supporting testing code in the documentation so that examples are always sure to compile.

One feature of Rust in particular that the framework developed for this thesis work heavily relies upon is the macro system [4] which is far more useful and powerful than the text substitution-based pre-processor type of macros available in other languages.



The past few years saw a notable increase in Rust's popularity, with companies like Microsoft that have invested millions into it and are actively porting substantial portions of their code bases to Rust. The Defense Advanced Research Projects Agency, also known as DARPA has stated their intention of converting legacy C code to Rust. In research, Rust won the Programming Languages Software Award in 2024 [5] given by ACM SIGPLAN with a citation that mentioned:

"… Born out of many years of research and experimentation, Rust tackles the real-world challenges and issues needed for practical adoption. Rust has been recognized as one of a handful of Safer Languages by NIST, and is increasingly deployed in industry for its safety benefits by large and small companies alike."

— sigplan.org/Awards/Software/2024_The_Rust_Programming_Language

In the annual developer survey, [6] carried out by Stackoverflow, Rust was elected the most admired language in 2024, and 83% of developers who use it want to continue using it.

3.1. The Macro System

What mainly pushed towards picking Rust as the language for this project is its native macro system which is extremely versatile and far more advanced than the macros one might be familiar with from C-like languages.

There are two types of macros in Rust, each for a specific purpose but in the end, what they all do is get code or tokens as input, perform code manipulation/expansion on it, and get the resulting code as output. Essentially macros allow us to use metaprogramming which is code that generates more code. A big reason for wanting metaprogramming in Rust is that macros are expanded at compile time and can have variadic parameters as well as implement traits which is something that functions cannot do since they get called at run time. The reduced amount of code needed to write when using macros is also an advantage for the maintainability of code bases in the long run.

3.1.1. Declarative Macros

```rust
// macro definition
macro_rules! power {
 ($value:expr, squared) => {{
 let x: u32 = $value;
 x.pow(2)
 }}
 ($value:expr, cubed) => {{
 let x: u32 = $value;
 x.pow(3)
  }}
 }
 // macro utilisation
 let square_of_2 = power!(2, squared);
 let cube_of_2 = power!(2, cubed);
```

Declarative macros [7] are invoked very much like a normal function gets invoked but they are marked with the ! symbol at the end of their name so there is no risk of mistaking a macro for a function or vice versa.

One example in addition to the one above is the println! macro which is available for the user's convenience and provides a very versatile way for printing to the standard output.

They are defined with macro_rules!, which is technically not part of Rust syntax but rather a syntax extension.

There must be at least one rule where each rule looks like the following:

```
($matcher) => {$expansion}
```

These kinds of macros can enormously reduce the amount of code to be written and are very frequently used in Rust. They are a great way to have less verbose code, however, in the scope of this thesis project there was no need for them therefore we shall no longer be concerned with them going forward.

Let us have a look at the other types of macros Rust possesses.

3.1.2. Procedural Macros

Procedural macros [8] are so named since they are nothing more than a procedure that gets code as input and outputs code, they are essentially a Rust function that generates other Rust code.

There is a further categorization of this kind of macro into three subtypes but at their core all three work almost the same with differences found in their inputs and outputs.

To write this kind of macros they must be defined in a separate crate of type proc-macro due to internal structure and name resolution reasons. All this complexity is probably going to be relaxed or even lifted in future versions of the language. One peculiarity of this type of crate is that it is enabled to export only macros and nothing else.

3.1.2.1. Function-Like Macros

```
// macro definition
#[proc_macro]
pub fn function_macro(input: TokenStream) -> TokenStream {
 TokenStream::new()
}
// macro utilization
function_macro!();
```

This first type of procedural macro can be invoked the way a declarative macro can. While at first glance they might seem identical to declarative macros they are not. Function-like macros are much more powerful than declarative macros since their input is not restricted in the same way that declarative macros inputs are, it can be anything and it can be transformed into anything within the bounds of the Rust language.

3.1.2.2. Derive Macros

```
// macro definition
#[proc_macro_derive(MyDerive)]
pub fn derive_macro(annotated_item: TokenStream) -> TokenStream {
 TokenStream::new()
}

// macro utilisation
#[derive(MyDerive)]
struct MyStrcut{
  // ...
 };
```

Derive Macros are also very commonly used in Rust since they are a very quick way of automatically implementing a trait on a struct or enum type. This is useful since Rust possesses quite a few traits such as Display, Debug, Clone, Copy, and Default just to name a few. Derive macros can also be used to automatically derive a user-defined trait which is why they are quite neat and spare programmers the tedious task of writing implementation blocks for each trait they want to have on a struct or enum by hand.

Once again, such macros are not used in this thesis work therefore we shall no longer be concerned with them going forward.

The last type of macro is what we will focus on and what our framework uses extensively.

3.1.2.3. Attribute Macros

```rust
// macro definition
#[proc_macro_attribute]
pub fn derive_macro(input: TokenStream, annotated_item: TokenStream) ->
TokenStream {
 TokenStream::new()
}

// macro utilization
#[derive_macro(attributes)]
fn example function(){
  // ...
 }

```

Derive macros are essentially used to add functionality to existing code by automatically implementing traits on them, however, they are restricted to structs and enums, most importantly they cannot add fields to an existing structure which is something we require. For these reasons, the type of macros used in cronframe are attribute macros which can be used on any kind of item and can also extensively alter the original code. In our specific use case, we will use attribute macros to parse structs only, since the parsing of enums is far more complex due to the various ways they can be used in Rust. This is however not a restriction because we can put structs inside enum variants and still make use of the framework functionalities.

3.2. Developing Macros

Developing macros requires parsing code into tokens and performing modifications or even turning text generated from this initial parsing into new tokens. All of this poses a big problem for anyone wanting to develop a macro: Rust is not a trivial language to parse.

Luckily some libraries make parsing the language much easier, some are also written specifically so that users can extend the language with macros according to their needs.

These libraries are the syn crate and the quote crate.

3.3. The Syn Crate

Available from crates.io, Rust's official crate registry:

```
```

```
$ cargo add syn
```

Syn [9] is a parsing library for parsing a stream of Rust tokens into a syntax tree of Rust source code geared toward use in Rust procedural macros.

Syn provides syntax trees starting from a syntax tree rooted at syn::File which represents a full source file of valid Rust source code to different entry points, such as:
- syn::Item
- syn::Expr
- syn::Type

Such entry points are of particular interest for macros.

Macros can have input parameters that need to be parsed themselves and support for that is also provided, for example in the case of Derive macros there is syn::DeriveInput which is any of the three legal input items to a derive macro.

Every token parsed by syn is associated with a span that tracks line and column information back to the source of that token, this allows the display of detailed error messages pointing to the causing source in the code.

An example of how to write a derive macro using syn is:

```rust
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput};

#[proc_macro_derive(MyMacro)]
pub fn my_macro(input: TokenStream) -> TokenStream {
 // parse the input tokens into a syntax tree
 let input = parse_macro_input!(input as DeriveInput);

  // build the output
  let expanded = quote! {
  // using quasi-quotation in this case
  };

  // hand the output tokens back to the compiler
  TokenStream::from(expanded)
 }
```

As we can see we get the input tokens and parse the input expecting valid syntax for the derive input specifically. Then we go on to generate the expanded code and finally return it as a token stream.

Now, there are a few ways to expand code:
- using the parse method implemented by syn on a string of text
- parse the input code and directly modify its syntax tree
- use quasi-quoting

The first one is the simplest by far, you just write code in a string and call the parse method but it might be a bother to inject data from the macro input into the string.

The second one is by far the most tedious of the three and probably the most error-prone approach as well since it requires to directly modify the syntax tree.

Finally, the last approach is a new thing the quote [10] crate allows us to do using the quote macro as per the example. With the quasi-quoting approach, you can write code normally inside the macro and inject variables from your macro expansion directly into the code that the macro needs to generate.

It is a bit of a "mix-world" where you use text and tokens at the same time to write code and it is the main approach used for this thesis.

3.4. The Quote Crate

The idea behind quasi-quoting is writing code that we treat as data.

Within the quote macro, we can write code that looks like the one we would write normally in a text editor and therefore we get all the benefits of using it such as brace matching, syntax highlighting, and indentation.

Autocompletion and type hinting are not always a given due to the complexity of the context the quote macro operates in but we might get it in some fortunate instances.

What differs from writing normal code to writing code inside the quote macro is that rather than compiling the code into the current crate, it is treated as data, passed around, and mutated until eventually it is handed back to the compiler as tokens to compile into the macro's crate.

An example of writing macros with the quote macro is:

```rust
extern crate proc_macro;

use proc_macro::TokenStream;
use quote::quote;

#[proc_macro_derive(MyMacro)]
pub fn my_macro(input: TokenStream) -> TokenStream {
 // parse the input and figure out what to do with it
 let name = {/* ... */};
  let expr = {/* ... */};

  // build the output
  let expanded = quote! {
  impl MyTrait for #name {
  fn my_trait_function(&self) -> usize {
  #expr
  }
  }
  };

  // hand the output tokens back to the compiler
  TokenStream::from(expanded)
 }
```

Here we use the content of the name and expr variables defined inside the body of the my_macro function called by the macro. What we do is use that content as data inside the quote macro so that we can implement the MyTrait trait on the content of the name variable which will be a type identifier and inside the function my_trait_function we use the content of the expr variable to define what the function should return.

This quasi-quoting approach is extremely useful as essentially it makes writing macros which are inherently complex pieces of code more like writing “normal” code.

4. The Framework

After an initial presentation of the framework is dealt with at the start of this chapter, what follows is an in-depth look at each component and how they work together. Finally, there are examples of macro expansions so to provide striking proof as to how much easier they make life for a user.

CronFrame is at its core a Rust framework that allows the definition of cron jobs from functions. An instance of the framework includes a web server that can be used to start and stop the scheduler, pause the scheduling of an individual job, and more. It also provides opt-in logging that is active by default but, should the user prefer to have their own, they can simply opt-out of it in the init function.

The framework allows for the definition of four types of jobs, all of these are functions but their context is different:

- **Global Jobs**: these are the jobs defined from basic Rust functions, those in the "global" scope, that is to say, they are not part of a type as they are standalone functions
- **Function Jobs**: in the context of struct types this first type of job is related to the type itself rather than to a concrete instance of it, meaning there will never be a duplicate of a function job on a type running in the scheduler
- **Method Jobs**: this second type of job that can be defined in struct types is related to an instance of the type as all methods are, at any given moment we can have multiple "versions" of the same job each related to a specific instance of the type they are defined on
- **CLI Jobs**: as the name implies these jobs are defined only through the CLI tool commands, they refer to external Rust code that is compiled and turned into a job

Examples of each job type follow.

```rust
// Global Job Example
#[cron(expr = "0/5 * * * * * *", timeout = "0")]
fn job_function() {
  println!("I print every 5 seconds...");
}

// Cron Object Example
#[cron_obj]
struct Greeting {
   name: String,
   expr: CronFrameExpr,
}

#[cron_impl]
impl Greeting {
   // Function Job Example
   #[fn_job(expr = "0/5 * * * * * *", timeout = "10000")]
   fn function_job() {
     println!("Have a good day!");
   }

   // Method Job Example
   #[mt_job(expr = "expr")]
```

```rust
    fn method_job(self) {
    println!("Hello {}", self.name);
  }
}

// CLI Job Example
fn main(){
  println!("the main function is the job function for CLI jobs!");
}
```

Notice the different contexts they are defined in. A CLI job is a peculiar case where the entire Rust code is the job and so main itself is to be considered as the job's function body.

Scheduling times are all UTC to keep macro attributes at a bare minimum, but for each job, a local time is provided on their web page.

Job recovery, much in line with the cron tool, is not a functionality that cronframe provides, at least at the time of writing. That is partly due to intricacies in the implementation of such functionality but also due to how it would impact the native macros-first intended utilization of the framework. How inflated macros would get if we had to mark a job either as recoverable or not, and how it should be recovered if more than one option is provided might too negatively impact user experience.

4.1. Framework Instance

A framework instance is required for jobs to be collected and parsed from their definitions and to be sent to the scheduler for execution. Such an instance carries a web server and a log configuration, both running right after completion of the init function while the scheduler will require invoking the start_scheduler method to get working.

The first thing the user will have to do is an initialization of the framework and then start the scheduler in the main function of their Rust project.

The initialization will also take care of gathering any global jobs and pass them to the scheduler which will set up their thread of execution exactly one second before they are supposed to run according to the cron expression dictated schedule.

For function and method jobs there are a few further steps to perform as it wouldn't make much sense to automatically collect them for two simple reasons:

- it is very subjective how and when a function job should be collected, if there is at least one instance of the type, or if a cron object simply serves as a job container
- method jobs are strictly related to a type instance which is possibly created well after the framework has been initialized

To front these problems, each cron object is injected with functions and methods that allow it to pass its jobs to a cronframe instance in a very straightforward way.

```rust
use cronframe::CronFrame;

fn main(){
  // framework initialization
```

```rust
    // global job collection happens here as well
    let cronframe = CronFrame::default();

    // scheduler start with keep main alive
    cronframe.run();
}
```

4.2. Annotation Macros

Macros are the main interface with the framework instance from the users' perspective and they have been the addressed focus point from the very start of development.

Job creation without macros is still supported but it is not the intended way of use for cronframe as a framework.

There is a total of five macros, specifically attribute macros as they are known in Rust:

- `cron`
- `cron_obj`
- `cron_impl`
- `fn_job`
- `mt_job`

The first one is a standalone macro to be used with "global scoped" functions to make cron jobs out of them. The other four macros instead are to be used with struct types, if we want to define cron jobs from methods or associated functions within them. Specifically, we need to use the cron_obj macro on top of a struct definition and the cron_impl macro on top of its implementation block which will only contain jobs for any given struct we want to use for cron job definition. The jobs contained in this implementation block will either be function or method jobs depending on which macro between fn_job and mt_job is used on top of the single function/method definition.

A struct type capable of hosting cron jobs inside it will be referred to as a **cron object** purely for ease of reference from here onwards.

These few macros allow us to generate code at compilation time therefore making all the required boilerplaty setup code a non-burden on the user so that they can turn their attention to the code that defines what the jobs do much sooner than it would take if the macros weren't there in the first place.

A more in-depth look and explanation of the macros follows.

4.2.1. Macro: cron

As mentioned before, this macro is used on top of standalone functions in Rust to define what is known in cronframe as "global cron jobs".

Such functions take no input arguments and return no value.

The macro itself takes two attributes which are:

- **expr**
  - the cron expression defining the scheduling for the job
- **timeout**
  - value in ms at which to stop the job after its first execution, it resets daily

The following code snippet shows how to use the macro:

```rust
#[cron(expr = "0 * * * * * *", timeout = "0")]
fn cron_job(){
  println!("I run every minute...");
}
```

The first thing done in the macro is the parsing of the attributes to check they are correct in name and quantity, expecting to get exactly two attributes named expr and timeout in that order. When this step has a positive outcome, the original function code will be parsed using the syn crate expecting to get an ItemFn. To this code, we will simply append a call to the submit macro of the inventory crate [11] in which we will create an instance of the JobBuilder type for the specific case of a global job. Using the inventory crate for global job collection allows us to collect all global job builders automatically at run time before even entering the main function so that when we init the framework we can easily get them and build the job instances to be used.

4.2.2. Macro(s) : cron_obj and cron_impl

These macros are only useful if both are used therefore they will be dealt with as if they were one. What they do is “turn” a struct into a cron object by acting on two different contexts of the struct with cron_obj acting on the struct definition, and cron_impl acting on an implementation block that is expected to solely contain job definitions.

Both macros take no attributes as they just set up a lot of framework code behind the scenes which is not dependent in any way on user directives.

A Cron Object is just a struct type that:
- automatically derives the **`Clone`** trait
- requires its fields to implement the Clone trait
- has a mutex boolean associated with it in the global scope
- has message-passing channels associated with it in the global scope
- has **two arrays of helper functions** returning builder instances associated with it in the global scope
  - one for functions jobs
  - one for method jobs
- can carry fields of type **`CronFrameExpr`** to be used to schedule cron jobs
- is injected with an additional field name **`tx`** for message passing between threads
- gets an implementation of the drop trait for dropping method jobs when their instance goes out of scope
- is injected with additional functions
  - `new_cron_obj` for creating a new instance of the cron object
  - `cf_drop_fn` for dropping function jobs
  - `cf_gather_fn` for passing all function jobs it contains to the cronframe instance
- is injected with additional methods
  - `cf_gather_mt` for passing all method jobs it contains to the cronframe instance
  - `cf_gather` for passing all the jobs it contains to the cronframe instance

The user can add as many fields as needed of type CronFrameExpr inside a cron object for job scheduling.

The following code snippet shows how to use the macros:

```rust
#[cron_obj]
struct User {
  expr1: CronFrameExpr,
  expr2: CronFrameExpr,
  name: String,
  age: u8,
}

#[cron_impl]
impl User {
  // define jobs here
}
```

Starting with the cron_obj macro, the first thing it does is inject a field named `tx` which is an `Option<Sender>` type from the crossbeam_channel crate used for message-passing.

Then it builds the definition of the `new_cron_obj` function which is variadic in number of parameters and has exactly as many as those defined by the user in the struct definition in the order they were defined, necessary to deal with the setup of the additional tx field.

To the edited structure it appends the mutex definition and the message-passing channels definition as well as the `Drop` trait implementation and an implementation block containing the `new_cron_obj` function and the `cf_drop_fn function`.

Last, but not least, it appends the definition of two arrays of helper functions that are annotated with a macro from the linkme [12] crate. This is what makes gathering them before main is executed possible. As soon as we step in the main function we can already instantiate a cron object and pass its jobs to the framework instance.

This macro is the first step for defining a struct type that can carry cron jobs and as mentioned above, it is not of much use by itself which is why after annotating a struct type with it, we need to annotate an implementation block with the `cron_impl` macro that will signal that every function or method contained in the block is expected to be a cron job.

The first thing the cron_impl macro does is fill the arrays defined by the cron_obj macro with the helper functions so that for each job that is inside the implementation block we add its helper function to the arrays.

The second thing this macro does is the addition of a new implementation block where the gathering functions are defined.

There are three types of gathering functions, all of them:
- build job instances from the `JobBuilders` returned by the helper functions
- set up the message-passing, and pass the jobs to the cronframe instance

In particular, **`cf_gather_fn`** is an associated function that takes care of function jobs, **`cf_gather_mt`** is a method that takes care of methods jobs and **`cf_gather`** is also a method that simply calls the previous two to handle both job types.

As we can see from the code example, inside the implementation block annotated with cron_impl is where we need to define jobs, which we can do with:

- **`fn_job`** macro
  - we use this macro on associated functions, therefore it defines a job for the type rather than one of its instances
- **`mt_job`** macro
  - we use this macro on methods, therefore it defines a job for an instance of the type

4.2.3. Macro: fn_job

This macro is the equivalent of the `cron` macro in the context of a cron object and as the cron macro does, it also takes two attributes which are:

- **expr**
  - the cron expression defining the scheduling for the job
- **timeout**
  - value in ms to stop the job after its first execution, it resets daily

This macro must be used in cron objects and only inside an implementation block annotated with the `cron_impl` macro.

It annotates an associated function turning it into a function job in the context of cronframe.

```rust
#[cron_obj]
struct User {
  expr1: CronFrameExpr,
  expr2: CronFrameExpr,
  name: String,
  age: u8,
}

#[cron_impl]
impl User {
  // function job definition
  #[fn_job(expr = "0 * * * * * *", timeout = "0")]
    fn my_function_job(){
    // ...
  }
}
```

The first thing this macro needs to do is make certain there is no `self` parameter in the function definition otherwise it would be meaningless to expand it since the context would be that of a method instead of an associated function.

Since the macro is the cron object equivalent of the `cron` macro, it also needs to check the number and the names of the attributes, asserting that there are only two named expr and timeout respectively. If that is the case it proceeds to add its helper function returning a `JobBuilder` instance for the job after the original function.

The helper function is named: **`cron_helper_<name of the original function>`**.

A macro allowing dead code is put on top of the original function definition since it might be falsely signaled as dead when it is being used through indirect calls by the cronframe instance. This might be lifted in the future as Rust analyzer or whatever tool in its stead becomes more able to track uses macro generated code, in the meanwhile this spares the user from annoying warnings.

4.2.4. Macro: mt_job

This macro is also similar to the `cron` macro in the context of a cron object but unlike the `cron` macro, it takes only one attribute:

- **expr**
  - the name of the CronFrameExpr field defined by the user for scheduling

The name of this attribute is the same as it was in the `cron` and `fn_job` macros however, this is not the cron expression in a string format. What this attribute expects is the name of a field of type `CronFrameExpr` defined by the user in the cron object structure.

There needs to be at least one of this type of field.

This macro must be used in cron objects and only inside an implementation block annotated with the `cron_impl` macro.

It annotates a method turning it into a method job in the context of cronframe.

```rust
#[cron_obj]
struct User {
  expr1: CronFrameExpr,
  expr2: CronFrameExpr,
  name: String,
  age: u8,
}

#[cron_impl]
impl User {
  // method job definition
  #[mt_job(expr = "expr1")]
    fn my_method_job(self){
    // ...
  }
}
```

This time around, the first thing the macro needs to do is make certain there is a `self` parameter in its definition otherwise it would be meaningless to expand it since the context would be that of an associated function instead of a method.

If there is a `self` parameter it goes on to check the presence of a single attribute named expr and, in that case, it generates:

- a helper function named **`cron_helper_<name of the original method>`**
- a function named **`cronframe_method_<name of the original method>`**

Both of these are added after the original definition of the method.

The field chosen for the expr attribute is injected into the construction of the helper function when parsing the cron expression. The last added function (`cronframe_method_<…>`) has the same code block as the original method but here any occurrence of self has been replaced with the cron object instance at the moment of job gathering.

This instance is then passed as an input parameter to the function as an `Any` type to be then down-cast to the type of the cron object. This means that essentially a copy of the cron object instance will be passed to the scheduler and they will evolve separately from that point forward.

4.3. The JobBuilder Type

A cron job gets built when invoking the `build` method from an instance of this type. `JobBuilder` instances are returned by the helper functions generated by the `fn_job` or the `mt_job` macros while with global jobs a `JobBuilder` is submitted to the inventory crate macro so it can be later obtained. Since what gets built can be either a global job, a function job, or a method job, this type is implemented with an enum containing struct variants.

```rust
#[derive(Debug, Clone)]
pub enum JobBuilder<'a> {
  Global {
    name: &'a str,
    job: fn(),
    cron_expr: &'a str,
    timeout: &'a str,
  },
  Method {
     name: &'a str,
     job: fn(arg: Arc<Box<dyn Any + Send + Sync>>),
     cron_expr: String,
     timeout: String,
     instance: Arc<Box<dyn Any + Send + Sync>>,
   },
   Function {
     name: &'a str,
     job: fn(),
     cron_expr: &'a str,
     timeout: &'a str,
   },
   CLI {
     name: &'a str,
     cron_expr: &'a str,
     timeout: &'a str,
   },
 }
```

As we can see, the Global and Function variants are virtually the same, their separation is only for keeping things in order from a logical standpoint as well as avoiding too much refactoring should their structure need changes in future versions. The Method variant is strikingly different in that it takes a more complex function pointer for the `job` field. It has a parameter that allows you to call it with a copy of the instance of the cron object hosted in the `instance` field.

The CLI variant is used for a type of job that can be added at run time from the CLI tool and will be dealt with in Chapter 5.

The implementation block of this type has three associated functions each returning an instance of one of the three variants, these are the functions that get called in the helper functions generated by the macros for them to return a `JobBuilder` instance.

Once we have an instance we require a way to build it which is exactly what the `build` method does. It first matches the type of job to build and then goes on to parse the cron expression and timeout. This is where each job gets its unique UUID and status channels. Life channels will be left to the `cf_gather` method to update as described earlier.

Once the build method returns, we end up with an instance of the `CronJob` type as described in the following paragraph.

The `JobBuilder` type is never used directly, any instance of it is automatically generated by macros or function calls on the framework instance.

4.4. The CronJob Type

This type contains what is required for a job to be scheduled to run by the scheduler and identified in the web server.

```rust
#[derive(Debug, Clone)]
pub struct CronJob {
  pub suspended: bool,
  pub name: String,
  pub id: Uuid,
  pub job: CronJobType,
  pub schedule: Schedule,
  pub timeout: Option<Duration>,
  pub timeout_notified: bool,
  pub status_channels: Option<(Sender<String>, Receiver<String>)>,
  pub life_channels: Option<(Sender<String>, Receiver<String>)>,
  pub start_time: Option<DateTime<Utc>>,
  pub run_id: Option<Uuid>,
  pub method_instance: Option<Arc<Box<dyn Any + Send + Sync>>>,
  pub failed: bool,
}
```

While fields like `name` and `id` are self-explanatory, others require some explanation.

The `suspended` field is a boolean that tells the scheduler whether the job is currently suspended from being scheduled or not. In case it is it simply gets skipped in the scheduler loop.

The `job` field is where the actual job resides as well as its type information.

The `schedule` field is a type obtained from parsing the cron expression which is used to handle future executions.

`Timeout` is an optional type containing a duration in milliseconds and the `timeout_notified` is a boolean required to log that a job has timed out.

Message-passing channels are used to handle job completion, abortion, and deletion and there are two different types. The `status_channels` handle all things concerning the job’s scheduling and execution events such as completion and abortion.

A job’s life, i.e. whether it is still alive and needs scheduling or it must be removed is handled by the `life_channels`. Though these only work for function and method jobs.

Removing global jobs is not available but should their execution no longer be required they can simply be suspended from scheduling.

The `start_time` field gets set upon the first schedule of the job and is used as the time base upon which timeout occurrence is checked. On timeout reset, it also gets reset when it will run once again.

Since jobs might fail on a specific run and execute just fine in others there needs to be a further way to identify a specific execution and this is what the `run_id` field is used for. It gets set anew each time a new execution of the job is required as per the `schedule` field.

The `method_instance` field is only set for method jobs and it is where the copy of the cron object to be used for running the job is hosted, without it we wouldn't be able to pass the context of the cron object to the scheduler.

Lastly, the `failed` field is simply a flag used to signal whether there have been any failed instances for a job in its execution history.

There are only methods associated with this type and as seen in previous examples, we never use it directly to build instances of it because that is what the `JobBuilder` type is meant to be used for.

The list of methods associated with the CronJob type is:
- `run`
- `run_graceful`
- `get_run_id`
- `type_to_string`
- `status`
- `schedule`
- `try_schedule`
- `set_schedule`
- `check_schedule`
- `set_timeout`
- `check_timeout`
- `timeout_reset`
- `timeout_to_string`
- `upcoming_utc`
- `upcoming_local`

The purpose of the methods is evident in their naming and most of them are only a few lines of code as shown in the appendix.

The core method involved in the scheduling is **`try_schedule`** which is invoked by the scheduler to see if an upcoming execution of the job is imminent by calling the `check_schedul` method that will return a true boolean value if the job is expected to run within the next second.

If that is the case, the job will be assigned a `run_id` for that specific execution and it will be scheduled for execution by calling the `run` method. Should the `run` method fail by returning any error, then the job will try to schedule for a limited period called graceful period which is by default 250ms but configurable by the user.

The `run` and `run_graceful` methods do nearly the same thing which is spawning two threads:
- control thread
- job thread

The **control thread** spawns the job_thread and it is required to check whether the job thread it spawned has terminated correctly or it aborted for whatever reason. This is done by waiting for the job thread to finish and checking its handle. If the handle contains an error a “JOB_ABORT” message is sent to the scheduler via status channels otherwise the job has been executed correctly and the message “JOB_COMPLETE” is sent to the scheduler.

The **job thread** does the actual execution of the job, it sleeps until the time for execution (which is within a second), and then it matches the type of the job to invoke the job function. If the job is a global job or a function job, the call to the job function is simply done without any input parameters as it doesn’t need any. For method jobs instead, we have to also get the instance of the cron object the job was created from which is hosted in the method_instance field, and pass it to the job function as input parameter for execution.

The difference between the two versions of the run method lies in the `run_graceful` method checking whether the upcoming execution is yet to come and in that case, waits for it but if that time already went by then it immediately executes the job. It also logs that a job has been executed during graceful time.

4.5. The CronFrame Type

An instance of this type represents the core of the framework.

Macros by themselves prepare all that is necessary for the jobs to be runnable but it is the `CronFrame` type that provides the execution.

```rust
pub struct CronFrame {
  pub cron_jobs: Mutex<Vec<CronJob>>,
  job_handles: Mutex<HashMap<String, JoinHandle<()>>>,
  _logger: Option<log4rs::Handle>,
  pub web_server_channels: (Sender<Shutdown>, Receiver<Shutdown>),
  pub filter: Option<CronFilter>,
  server_handle: Mutex<Option<Shutdown>>,
  pub quit: Mutex<bool>,
  pub grace: u32,
   pub running: Mutex<bool>,
}
```

In the type definition, we find a mutex of a vector of `CronJobs` which is where all the jobs are collected, the job pool at the scheduler’s disposal. For each job that is in a running state, we have a thread join handle stored in a hashmap with the job’s id as the key.

The logger handle is wrapped in an `Option` type since the user might want their own tailored configuration for logging instead of using the default one provided by the framework.

Webserver channels are used for the setup and start-up of the web server when calling the `init` associated function which takes as arguments an enum variant wrapped in an `Option` for the filter and a boolean for whether or not to use the default logger.

The filter allows the scheduler to run only one type of job among global, function, method, or CLI jobs should the user want to do so.

For the user’s convenience, there is also a `default` associated function that takes no arguments and calls in its body the init function with no filter and the default logger configuration enabled.

The `grace` field contains either the default value of 250ms for the graceful period or the one set by the user in the configuration file.

The `running` mutex boolean simply signals whether the scheduler is currently running or not. The `quit` mutex boolean is used for terminating the cronframe instance.

The `init` function first sets up the logger, creates an instance of Cronframe with no jobs, and starts looking for global job builders that have been collected with the `cron` macro using the inventory crate, then it calls the `build` method and gets the jobs built and pushes them into the vector of jobs.

The initialized instance of cronframe is now ready but before it can be passed to the server it needs to be wrapped in an `Arc` type since it will be shared by the main thread of the program as well as the web server thread.

The web server thread is started with an `Arc` instance of the cronframe framework and we wait for its thread to pass via message passing the quit handle for the server to the cronframe instance. Once we have it we can return an `Arc` instance of cronframe where the `init` function was invoked.

At this point, everything is set up and the server is running but the scheduler is not.

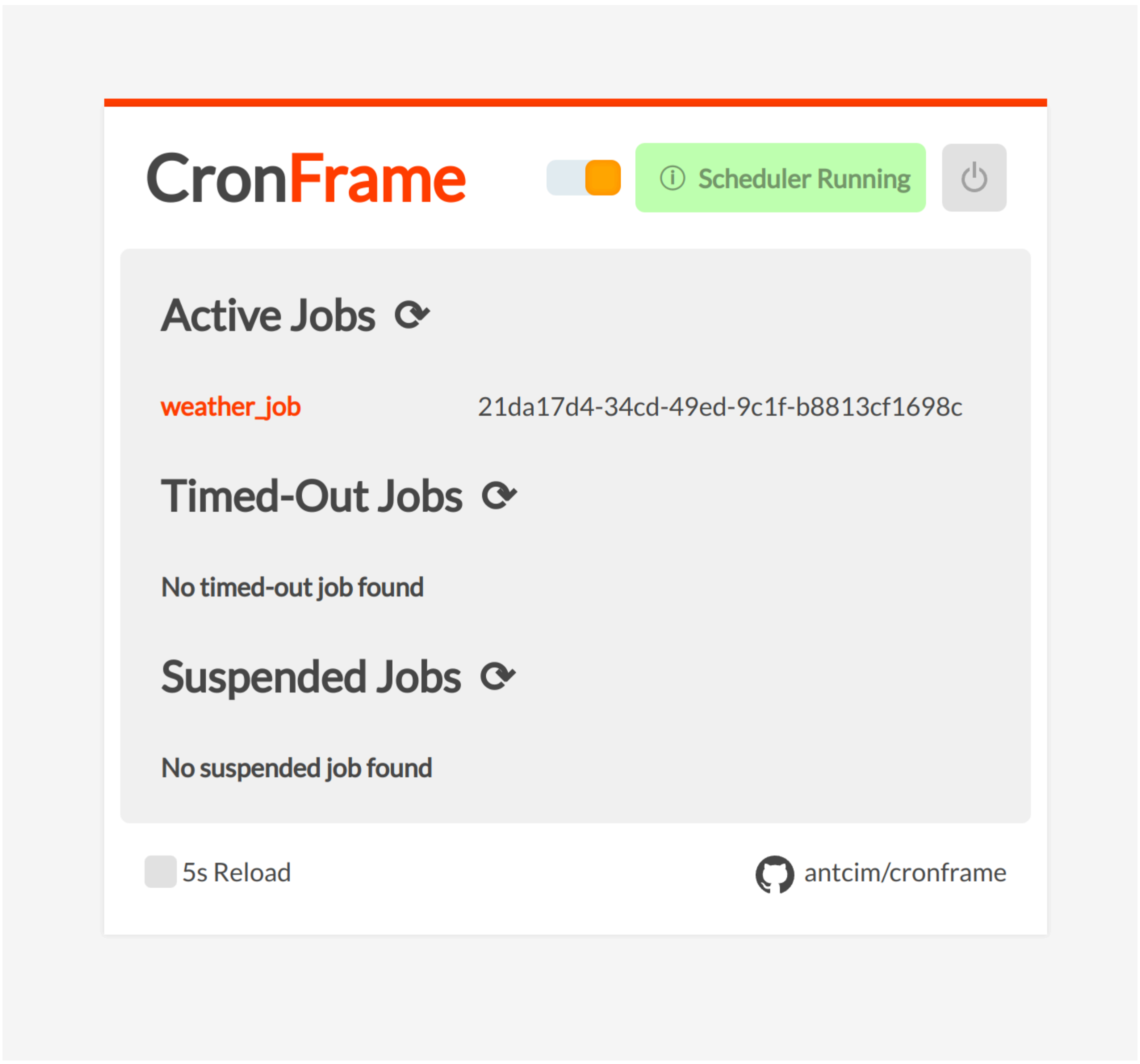
We just need to call the `start_scheduler` method to start it and bear in mind that this will not keep the main thread alive. To keep it alive a user either implements their program logic or calls the `keep_alive` method which simply runs an infinite loop that does nothing but sleep for 500ms and checks whether a quit request has been filed. In case it has, it breaks the loop terminating the framework instance.

A `run` method that starts the scheduler and keeps it alive is also provided for convenience.

The initialization process only collects global jobs since that is all that is known at compile time before anything in the main function is executed. For adding function jobs and method jobs to the scheduler, the Cronframe type provides an `add_job` method which is used in the `cf_gather`, `cf_gather_fn`, and `cf_gather_mt` functions available in cron objects.

Should the user not want to use macros, there is a `new_job` method that can be used to add jobs to the cronframe instance from either global functions or closures. These jobs will be typed as global jobs.

4.6. The Web Server: Rocket



Active Jobs

weather_job 21da17d4-34cd-49ed-9c1f-b8813cf1698c

Timed-Out Jobs

No timed-out job found

Suspended Jobs

No suspended job found

Figure 2: Home Page of the Webserver

The user might want to start and stop the scheduler, see the status of the jobs in the job pool, and check whether they are running, awaiting for schedule or if they have timed out, or maybe make sure the job hasn't failed running instances in the past.

A user might also want to change the scheduling or timeout and they might want to do all this with a GUI when their application is running. For these purposes, a web server is at the user's disposal.

By default, the server is set to run on port 8098 at the address 127.0.0.1 also known as localhost.

The ip and port can be changed with a configuration file as the pertaining paragraph will later show.

The web server uses the Rocket framework [13] since it provides a well-tested multithreaded runtime environment as well as common functionality and a very configurable setup to get a web server up and running without too much boilerplate code.

Upon the first start of the framework inside the current directory, a templates directory will be generated containing the following files:
- base.html.tera

- index.html.tera
- job.html.tera
- tingle.js
- cronframe.js
- styles.css
- tingle.css

The .tera files contain common HTML extended with templating capabilities parsed by the tera engine provided with Rocket. The .js files are tingle.js which is from a modal library and cronframe.js made for the web server functionality. The style files are there to give a somewhat pleasing appearance to the interface with support of light and dark themes.

The templates directory allows for custom configuration of the web server appearance should the user feel the need to.

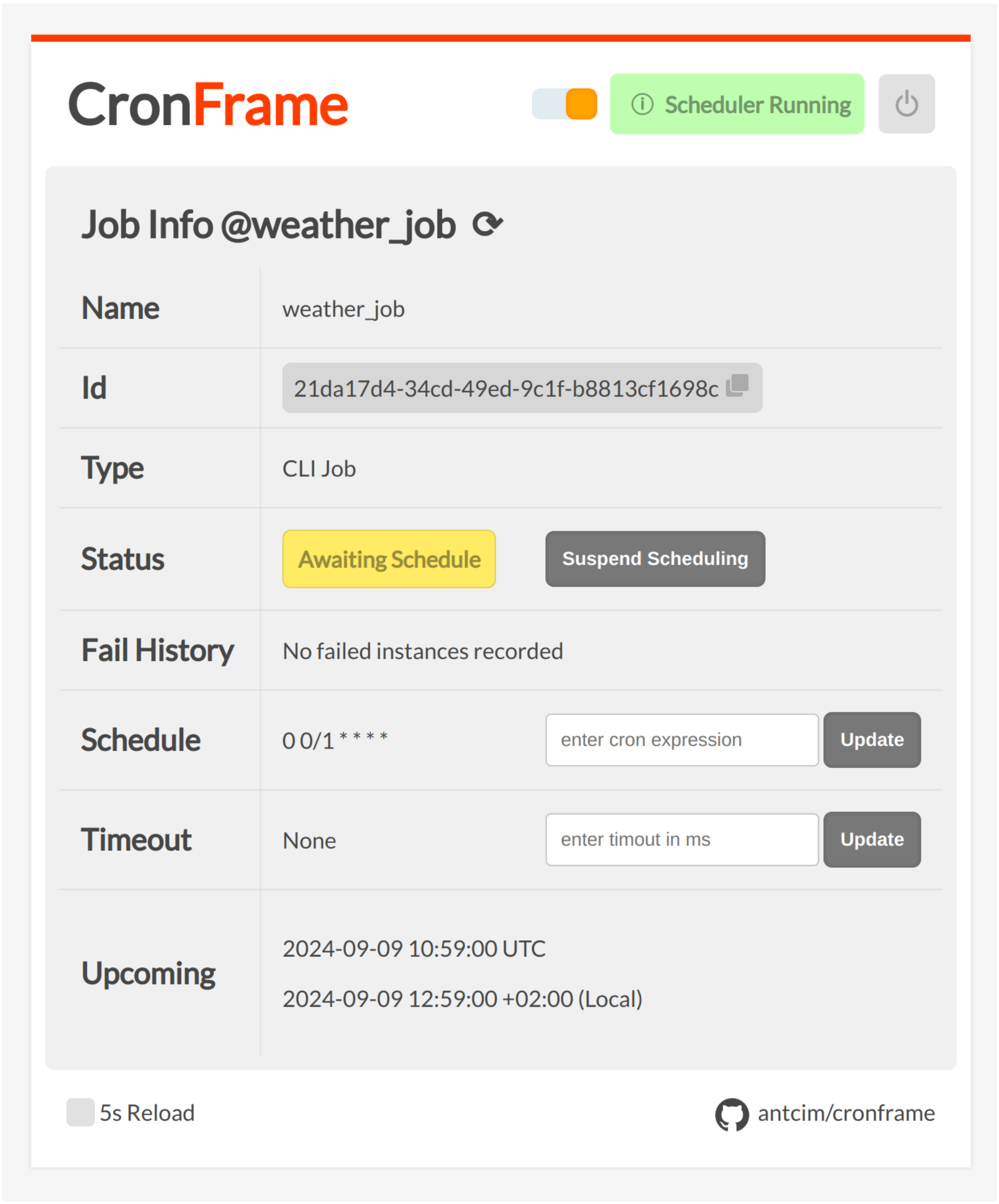
There is a total of nine routes provided by the server:

- **job list route**
  - found at http://127.0.0.1:8098/
  - it shows a list of jobs in the current pool with their IDs, clicking on a job name redirects to the job's info route
- **job info route**
  - http://127.0.0.1:8098/job/<name>/<id>
  - it provides a page with the job's definition and status information
- **schedule set API route**
  - found at http://127.0.0.1:8098/job/<name>/<id>/schedset/<expression>
  - it provides a way to change the schedule of a job and is used in the job info route
- **timeout set API route**
  - found at http://127.0.0.1:8098/job/<name>/<id>/toutset/<value>
  - it provides a way to change the timeout of a job and is used in the job info route
- **job suspension toggle API route**
  - found at http://127.0.0.1:8098/job/<name>/<id>/suspension_toggle
  - it allows to suspend or reprise of the scheduling of a single job
- **scheduler start API route**
  - found at http://127.0.0.1:8098/start_scheduler/
- **scheduler stop API route**
  - found at http://127.0.0.1:8098/stop_scheduler/
- **add CLI job API route**
  - found at http://127.0.0.1:8098/add_cli_job/<expr>/<timeout>/<job>
- **cronframe shutdown API route**
  - found at http://127.0.0.1:8098/shutdown

Each route is checked against a route guard that in addition to providing the requested route also provides type checking for dynamic components in the route, that is to say, the parts enclosed by angle brackets.

All dynamic components in the above routes are regarded to be of type string except for the <value> component of the timeout set route which is checked to be compatible with the i64 type.

Every route carries additional hidden data, this data is the Arc of the cronframe instance that has been passed to the web server thread during the init of the framework.


CronFrame

Job Info @weather_job

Name	weather_job
Id	21da17d4-34cd-49ed-9c1f-b8813cf1698c
Type	CLI Job
Status	Awaiting Schedule
Fail History	No failed instances recorded
Schedule	0 0/1 * * * *
Timeout	None
Upcoming	2024-09-09 10:59:00 UTC 2024-09-09 12:59:00 +02:00 (Local)

Figure 3: Job Page on the Webserver

4.7. The Logger: log4rs

The logger is implemented with the log4rs [14] crate which is itself affine to the log4j library.

Logging cron jobs can lead to dealing with enormous file sizes depending on the scheduling the jobs have. For this reason, instead of using a logger that always appends to a file, the default logger is set up in a rolling logger configuration with 3 files of archive.

Once the latest.log file reaches 1MB of maximum size it becomes an archive file and a new latest file is written. Archive files are named archive_i.log where i is a digit that goes from 0 to archive_files - 1, with 0 being the more recent file of the archives.

The logger is extensively configurable, it can be entirely not used or its parameters can be configured from a configuration file, such as:

- dir
  - the directory where to store log files
- file_size
  - maximum file size in megabytes of the log files
- archive_files
  - the number of archive files to generate
- latest_file_name
  - the name to be used for the current log file
- archive_file_name
  - the name to be used for the archive files
- msg_pattern
  - the message pattern for what to print when logging
- level_filter
  - the log message filter for what to write on the log file

4.8. The Configuration File

Rust's package manager Cargo makes extensive use of toml files for configuration purposes. Each Rust project has a Cargo.toml where package configuration, dependencies, workspace configuration, development dependencies, and other types of configurations are stored.

This allows for a clear overview and edit of any configuration available in that file which is fast and painless for the user to do.

Other rust crates, for example, Rocket itself, make use of additional toml files to aid user configurability.

For this reason a cronframe.toml configuration file can be used with the framework to set a few things when it comes to the server, the logger, and the scheduler.

The following is an example of a configuration file with all available fields set to the default configuration:

```toml
[webserver]
port = 8098
ip = "127.0.0.1"

[logger]
dir = "log"
file_size = 1 # this is in MB
archive_files = 3
latest_file_name = "latest"
archive_file_name = "archive"
msg_pattern = "{d(%Y-%m-%d %H:%M:%S %Z)} {l} {t} - {m}{n}"
level_filter = "info"

[scheduler]
grace = 250 # this is in ms
```

As we can see there are three sections marked by square brackets, one for the webserver, one for the logger, and one for the scheduler. Each of these sections is optional and each field in a section is optional as well.

In the web server section we can set the:
- ip
  - the ip to use to run the server
  - 127.0.0.1 (also localhost) by default
- port
  - the port number where the webserver will be available at
  - 8098 by default

In the logger section we can set:
- dir
  - the directory to use to save the log files
  - "log" by default
- file_size
  - the size in megabytes of a single log file
  - 1MB by default
- archive_files
  - the number of archive files to retain in addition to the latest file
  - 3 by default
- latest_file_name
  - the file name for the latest log file
  - "latest" by default
- archive_file_name
  - the file name for the archive files which will end in a number automatically
  - "archive" by default
- msg_pattern
  - the pattern to use when logging a message on a file
  - "{d(%Y-%m-%d %H:%M:%S %Z)} {l} {t} - {m}{n}" by default
  - refer to the log crate on crates.io for message syntax
- level_filter
  - the level filter to use when logging
  - possible choices are: "error", "off", "warn", "debug"
  - "info" by default

In the scheduler section we can set:
- grace
  - the graceful period to be used for job scheduling
  - 250 milliseconds by default

The full code of all framework components we have seen is available in the Appendix.

4.9. Macro Expansions

The rest of this chapter contains examples of the code produced by each macro expansion to show how much the macro themselves prepare the field for the utilization of the framework.

4.9.1. Macro Expansion: cron

The code written by the user is the following:

```rust
#[macro_use]
extern crate cronframe;

use cronframe::{CronFrame, CronFrameExpr};

#[cron(expr="0 0/30 * * * * *", timeout="0")]
fn my_global_job(){
  // job code here...
}
fn main() {
  let cronframe = CronFrame::default();
  cronframe.run();
}
```

We have a global job that doesn't do anything but the code inside its body is not relevant to the purpose of showing the expansion of the macro.

At compile time these few lines of code will be expanded in the following according to what has been described earlier:

```rust
#[macro_use]
extern crate cronframe;
use cronframe::{CronFrame, CronFrameExpr};
fn my_global_job() {}
#[allow(non_upper_case_globals)]
const _: () = {
  static __INVENTORY: ::inventory::Node = ::inventory::Node {
    value: &{
    cronframe::JobBuilder::global_job(
    "my_global_job",
    my_global_job,
    "0 0/30 * * * * *",
    "0",
    )
  },
  next: ::inventory::core::cell::UnsafeCell::new(
  ::inventory::core::option::Option::None,
  ),
  };
  #[link_section = ".text.startup"]
  unsafe extern "C" fn __ctor() {
    unsafe { ::inventory::ErasedNode::submit(__INVENTORY.value, &__INVENTORY) }
  }
  #[used]
  #[link_section = ".init_array"]
  static __CTOR: unsafe extern "C" fn() = __ctor;
};
fn main() {
  let cronframe = CronFrame::default();
  cronframe.run();
}
```

Immediately, the amount of work that macros spare us is quite astonishing. We can see that an instance of a JobBuilder for a global job is constructed and collected by the inventory crate right on top of main so that when default is called it will have the JobBuilder at its disposal by calling the inventory crate's functionality.

4.9.2. Macro Expansion: cron_obj, cron_impl, fn_job, mt_job

This following expansion example is quite big, first of all as we know, the cron_obj macro and the cron_impl macro are just one macro split in two due to the context of the operation. Also, neither of these two macros is useful if we do not define a job at the end, therefore we define both types of cron objects jobs in the same implementation block to show what comes out of a full-fledged cron object.

The code written by the user is the following:

```rust
#[macro_use]
extern crate cronframe;

use cronframe::{CronFrame, CronFrameExpr};

#[cron_obj]
struct ExpansionObj{
  cron_expr: CronFrameExpr,
}

#[cron_impl]
impl ExpansionObj{
  #[fn_job(expr="0 0/30 * * * * *", timeout="0")]
    fn my_function_job(){
    // my function job code here...
  }

  #[mt_job(expr="cron_expr")]
    fn my_method_job(self){
    // my method job code here...
  }
}

fn main() {
  let cronframe = CronFrame::default();
  cronframe.run();
}
```

What these few lines of code will become is the following:

```rust
#[macro_use]
extern crate cronframe;
```

```rust

use cronframe::{CronFrame, CronFrameExpr};

#[cron_obj]
struct ExpansionObj{
    cron_expr: CronFrameExpr,
}

#[cron_impl]
impl ExpansionObj{
    #[fn_job(expr="0 0/30 * * * *", timeout="0")]
    fn my_function_job(){
        // my function job code here...
    }

    #[mt_job(expr="cron_expr")]
    fn my_method_job(self){
        // my method job code here...
    }
}

fn main() {
    let cronframe = CronFrame::default();
    cronframe.run();
}
```

What these few lines of code will become is the following:

```rust
#[macro_use]
extern crate cronframe;
use cronframe::{CronFrame, CronFrameExpr};
struct ExpansionObj {
    cron_expr: CronFrameExpr,
    tx: Option<cronframe::Sender<String>>,
}
#[automatically_derived]
impl ::core::clone::Clone for ExpansionObj {
    #[inline]
    fn clone(&self) -> ExpansionObj {
        ExpansionObj {
            cron_expr: ::core::clone::Clone::clone(&self.cron_expr),
            tx: ::core::clone::Clone::clone(&self.tx),
        }
    }
}
static CF_FN_JOBS_FLAG_EXPANSIONOBJ: std::sync::Mutex<bool> =
std::sync::Mutex::new(
    false,
);
static CF_FN_JOBS_CHANNELS_EXPANSIONOBJ: cronframe::Lazy<
    (cronframe::Sender<String>, cronframe::Receiver<String>),
> = cronframe::Lazy::new(|| cronframe::bounded(1));
```

```rust
impl Drop for ExpansionObj {
    fn drop(&mut self) {
        if self.tx.is_some() {
            let _ =
self.tx.as_ref().unwrap().send("JOB_DROP".to_string());
        }
    }
}
impl ExpansionObj {
    fn new_cron_obj(cron_expr: CronFrameExpr) -> ExpansionObj {
        ExpansionObj {
            cron_expr,
            tx: None,
        }
    }
    fn cf_drop(&self) {
        if *CF_FN_JOBS_FLAG_EXPANSIONOBJ.lock().unwrap() {
            for func in CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ {
                let _ =
CF_FN_JOBS_CHANNELS_EXPANSIONOBJ.0.send("JOB_DROP".to_string());
            }
            *CF_FN_JOBS_FLAG_EXPANSIONOBJ.lock().unwrap() = false;
        }
    }
}
static CRONFRAME_METHOD_JOBS_EXPANSIONOBJ: ::linkme::DistributedSlice<
    [fn(
        std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
    ) -> cronframe::JobBuilder<'static>],
> = {
    #[cfg(
        any(
            target_os = "none",
            target_os = "linux",
            target_os = "macos",
            target_os = "ios",
            target_os = "tvos",
            target_os = "android",
            target_os = "fuchsia",
            target_os = "illumos",
            target_os = "freebsd",
            target_os = "openbsd",
            target_os = "psp",
        )
    )]
    extern "Rust" {
        #[link_name =
"__start_linkme_CRONFRAME_METHOD_JOBS_EXPANSIONOBJ"]
        static LINKME_START: <[fn(
            std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
        ) -> cronframe::JobBuilder<'static>]
as ::linkme::__private::Slice>::Element;
        #[link_name = "__stop_linkme_CRONFRAME_METHOD_JOBS_EXPANSIONOBJ"]
        static LINKME_STOP: <[fn(
            std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
        ) -> cronframe::JobBuilder<'static>]
as ::linkme::__private::Slice>::Element;
        #[link_name =
"__start_linkm2_CRONFRAME_METHOD_JOBS_EXPANSIONOBJ"]
        static DUPCHECK_START: ::linkme::__private::usize;
        #[link_name = "__stop_linkm2_CRONFRAME_METHOD_JOBS_EXPANSIONOBJ"]
        static DUPCHECK_STOP: ::linkme::__private::usize;
    }
    #[used]
    #[cfg(
        any(
            target_os = "none",
            target_os = "linux",
            target_os = "android",
            target_os = "fuchsia",
            target_os = "illumos",
            target_os = "freebsd",
            target_os = "openbsd",
            target_os = "psp",
        )
    )]
    #[link_section = "linkme_CRONFRAME_METHOD_JOBS_EXPANSIONOBJ"]
    static mut LINKME_PLEASE: [<[fn(
        std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
    ) -> cronframe::JobBuilder<'static>]
as ::linkme::__private::Slice>::Element; 0] = [];
    #[used]
    #[link_section = "linkm2_CRONFRAME_METHOD_JOBS_EXPANSIONOBJ"]
    static DUPCHECK: ::linkme::__private::usize = 1;
    if !(::linkme::__private::mem::size_of::<
        <[fn(
            std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
        ) -> cronframe::JobBuilder<'static>]
as ::linkme::__private::Slice>::Element,
    >() > 0)
    {
        ::core::panicking::panic(
            "assertion
failed: ::linkme::__private::mem::size_of::<<[fn(std::sync::Arc<Box<dyn
std::any::Any +\n                     Send + Sync>>) ->
cronframe::JobBuilder<\'static>]
as\n            ::linkme::__private::Slice>::Element>() > 0",
        )
    }
    unsafe {
        ::linkme::DistributedSlice::private_new(
            "CRONFRAME_METHOD_JOBS_EXPANSIONOBJ",
            &raw const LINKME_START,
            &raw const LINKME_STOP,
            &raw const DUPCHECK_START,
            &raw const DUPCHECK_STOP,
```

```rust
        )
    }
};
#[doc(hidden)]
use _linkme_macro_CRONFRAME_METHOD_JOBS_EXPANSIONOBJ as
CRONFRAME_METHOD_JOBS_EXPANSIONOBJ;
static CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ: ::linkme::DistributedSlice<
    [fn() -> cronframe::JobBuilder<'static>],
> = {
    #[cfg(
        any(
            target_os = "none",
            target_os = "linux",
            target_os = "macos",
            target_os = "ios",
            target_os = "tvos",
            target_os = "android",
            target_os = "fuchsia",
            target_os = "illumos",
            target_os = "freebsd",
            target_os = "openbsd",
            target_os = "psp",
        )
    )]
    extern "Rust" {
        #[link_name =
"__start_linkme_CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ"]
        static LINKME_START: <[fn() -> cronframe::JobBuilder<
            'static,
        >] as ::linkme::__private::Slice>::Element;
        #[link_name =
"__stop_linkme_CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ"]
        static LINKME_STOP: <[fn() -> cronframe::JobBuilder<
            'static,
        >] as ::linkme::__private::Slice>::Element;
        #[link_name =
"__start_linkm2_CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ"]
        static DUPCHECK_START: ::linkme::__private::usize;
        #[link_name =
"__stop_linkm2_CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ"]
        static DUPCHECK_STOP: ::linkme::__private::usize;
    }
    #[used]
    #[cfg(
        any(
            target_os = "none",
            target_os = "linux",
            target_os = "android",
            target_os = "fuchsia",
            target_os = "illumos",
            target_os = "freebsd",
            target_os = "openbsd",
            target_os = "psp",
        )
    )]
    #[link_section = "linkme_CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ"]
    static mut LINKME_PLEASE: [<[fn() -> cronframe::JobBuilder<
        'static,
    >] as ::linkme::__private::Slice>::Element; 0] = [];
    #[used]
    #[link_section = "linkm2_CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ"]
    static DUPCHECK: ::linkme::__private::usize = 1;
    if !(::linkme::__private::mem::size_of::<
        <[fn() -> cronframe::JobBuilder<'static>] as ::linkme::__private::Slice>::Element,
    >() > 0)
    {
        ::core::panicking::panic(
            "assertion failed: ::linkme::__private::mem::size_of::<<[fn() -> cronframe::JobBuilder<\'static>]\n as ::linkme::__private::Slice>::Element>() > 0",
        )
    }
    unsafe {
        ::linkme::DistributedSlice::private_new(
            "CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ",
            &raw const LINKME_START,
            &raw const LINKME_STOP,
            &raw const DUPCHECK_START,
            &raw const DUPCHECK_STOP,
        )
    }
};
#[doc(hidden)]
use _linkme_macro_CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ as CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ;
impl ExpansionObj {
    #[allow(dead_code)]
    fn my_function_job() {}
    fn cron_helper_my_function_job() -> cronframe::JobBuilder<'static> {
        cronframe::JobBuilder::function_job(
            "my_function_job",
            Self::my_function_job,
            "0 0/30 * * * * *",
            "0",
        )
    }
    #[allow(dead_code)]
    fn my_method_job(self) {}
    fn cron_method_my_method_job(
        arg: std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
    ) {
        let cron_frame_instance = arg.clone();
        let cronframe_self = (*cron_frame_instance).downcast_ref::<Self>().unwrap();
    }
    fn cron_helper_my_method_job(
        arg: std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
    ) -> cronframe::JobBuilder<'static> {
        let instance = arg.clone();
        let this_obj = (*instance).downcast_ref::<Self>().unwrap();
        let expr = this_obj.cron_expr.expr();
        let tout = {
            let res = ::alloc::fmt::format(
                format_args!("{0}", this_obj.cron_expr.timeout()),
            );
            res
        };
        let instance = arg.clone();
        cronframe::JobBuilder::method_job(
            "my_method_job",
            Self::cron_method_my_method_job,
            expr.clone(),
            tout,
            instance,
        )
    }
}
#[used]
#[link_section = "linkme_CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ"]
static LINKME_MY_FUNCTION_JOB_0: fn() -> cronframe::JobBuilder<'static>
= {
    #[allow(clippy::no_effect_underscore_binding)]
    unsafe fn __typecheck(_: ::linkme::__private::Void) {
        let __new = || -> fn() -> &'static fn() ->
cronframe::JobBuilder<'static> {
            || &LINKME_MY_FUNCTION_JOB_0
        };
        unsafe {
            ::linkme::DistributedSlice::private_typecheck(
                CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ,
                __new(),
            );
        }
    }
    ExpansionObj::cron_helper_my_function_job
};
#[used]
#[link_section = "linkme_CRONFRAME_METHOD_JOBS_EXPANSIONOBJ"]
static LINKME_MY_METHOD_JOB_1: fn(
    _self: std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
) -> cronframe::JobBuilder<'static> = {
    #[allow(clippy::no_effect_underscore_binding)]
    unsafe fn __typecheck(_: ::linkme::__private::Void) {
        let __new = || -> fn() -> &'static fn(
            _self: std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>,
        ) -> cronframe::JobBuilder<'static> { ||
&LINKME_MY_METHOD_JOB_1 };
```

```
        unsafe {
            ::linkme::DistributedSlice::private_typecheck(
                CRONFRAME_METHOD_JOBS_EXPANSIONOBJ,
                __new(),
            );
        }
    }
    ExpansionObj::cron_helper_my_method_job
};
impl ExpansionObj {
    pub fn cf_gather_mt(&mut self, frame: std::sync::Arc<CronFrame>) {
        {
            let lvl = ::log::Level::Info;
            if lvl <= ::log::STATIC_MAX_LEVEL && lvl <= ::log::max_level() {
                ::log::__private_api::log(
                    format_args!("Collecting Method Jobs from {0}", "EXPANSIONOBJ"),
                    lvl,
                    &("expansions", "expansions", ::log::__private_api::loc()),
                    (),
                );
            }
        };
        if !CRONFRAME_METHOD_JOBS_EXPANSIONOBJ.is_empty() {
            let life_channels = cronframe::bounded(1);
            self.tx = Some(life_channels.0.clone());
            for method_job in CRONFRAME_METHOD_JOBS_EXPANSIONOBJ {
                let job_builder = (method_job)(
                    std::sync::Arc::new(Box::new(self.clone())),
                );
                let mut cron_job = job_builder.build();
                cron_job.life_channels = Some(life_channels.clone());
                {
                    let lvl = ::log::Level::Info;
                    if lvl <= ::log::STATIC_MAX_LEVEL && lvl <= ::log::max_level() {
                        ::log::__private_api::log(
                            format_args!(
                                "Found Method Job \"{0}\" from {1}.",
                                cron_job.name,
                                "EXPANSIONOBJ",
                            ),
                            lvl,
                            &("expansions", "expansions", ::log::__private_api::loc()),
                            (),
                        );
                    }
                };
                frame.clone().add_job(cron_job);
            }
            {
                let lvl = ::log::Level::Info;
                if lvl <= ::log::STATIC_MAX_LEVEL && lvl <= ::log::max_level() {
                    ::log::__private_api::log(
                        format_args!("Method Jobs from {0} Collected.", "EXPANSIONOBJ"),
                        lvl,
                        &("expansions", "expansions", ::log::__private_api::loc()),
                        (),
                    );
                }
            };
        } else {
            {
                let lvl = ::log::Level::Info;
                if lvl <= ::log::STATIC_MAX_LEVEL && lvl <= ::log::max_level() {
                    ::log::__private_api::log(
                        format_args!(
                            "Not Method Jobs from {0} has been found.",
                            "EXPANSIONOBJ",
                        ),
                        lvl,
                        &("expansions", "expansions", ::log::__private_api::loc()),
                        (),
                    );
                }
            };
        }
    }
    pub fn cf_gather_fn(frame: std::sync::Arc<CronFrame>) {
        {
            let lvl = ::log::Level::Info;
            if lvl <= ::log::STATIC_MAX_LEVEL && lvl <= ::log::max_level() {
                ::log::__private_api::log(
                    format_args!("Collecting Function Jobs from {0}", "EXPANSIONOBJ"),
                    lvl,
                    &("expansions", "expansions", ::log::__private_api::loc()),
                    (),
                );
            }
        };
        if !CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ.is_empty() {
            let fn_flag = *CF_FN_JOBS_FLAG_EXPANSIONOBJ.lock().unwrap();
            if !fn_flag {
                for function_job in CRONFRAME_FUNCTION_JOBS_EXPANSIONOBJ {
                    let job_builder = (function_job)();
                    let mut cron_job = job_builder.build();
                    cron_job.life_channels = Some(
                        CF_FN_JOBS_CHANNELS_EXPANSIONOBJ.clone(),
                    );
                    {
                        let lvl = ::log::Level::Info;
                        if lvl <= ::log::STATIC_MAX_LEVEL && lvl <= ::log::max_level() {
                            ::log::__private_api::log(
                                format_args!(
                                    "Found Function Job \"{0}\" from {1}.",
                                    cron_job.name,
                                    "EXPANSIONOBJ",
                                ),
                                lvl,
                                &("expansions", "expansions", ::log::__private_api::loc()),
                                (),
                            );
                        }
                    };
                    frame.clone().add_job(cron_job);
                }
                {
                    let lvl = ::log::Level::Info;
                    if lvl <= ::log::STATIC_MAX_LEVEL && lvl <= ::log::max_level() {
                        ::log::__private_api::log(
                            format_args!(
                                "Function Jobs from {0} Collected.",
                                "EXPANSIONOBJ",
                            ),
                            lvl,
                            &("expansions", "expansions", ::log::__private_api::loc()),
                            (),
                        );
                    }
                };
                *CF_FN_JOBS_FLAG_EXPANSIONOBJ.lock().unwrap() = true;
            }
        } else {
            {
                let lvl = ::log::Level::Info;
                if lvl <= ::log::STATIC_MAX_LEVEL && lvl <= ::log::max_level() {
                    ::log::__private_api::log(
                        format_args!(
                            "Not Function Jobs from {0} has been found.",
                            "EXPANSIONOBJ",
                        ),
                        lvl,
                        &("expansions", "expansions", ::log::__private_api::loc()),
                        (),
                    );
                }
            };
        }
    }
    pub fn cf_gather(&mut self, frame: std::sync::Arc<CronFrame>) {
        self.cf_gather_mt(frame.clone());
        Self::cf_gather_fn(frame.clone());
    }
}
```

Macro expansion of a cron object generates a lot more code, even more so due to the expansion of the linkme crate which is necessary to the functioning of cronframe.

If we have a look, amid all that complicated code we will find the arrays of helper functions, the implementation blocks containing the gathering functions, the mutexes and message-passing channels, and all that has been described in the macro paragraphs.

5. The CLI Tool

In this chapter, we will deal with how the CLI tool came about, what it allows a user to do, and list all of its commands. There is also a chapter dedicated to general information about the tool to convey how it works and why it does things in a certain way.

The birth of the CLI tool came about to address the need of users who might not want a framework inside their code base and would much rather use a cron job scheduler program with the capability of adding jobs to it with little to no effort.

The CLI tool provides a global instance of cronframe which sets up its configuration and templating files in the .cronframe directory under the user directory when it is run for the first time.

Log files will also be found inside the log directory under the .cronframe directory.

Since Rust allows for the definition of a library and a binary inside the same crate, the same project can work for both the framework and the CLI tool which makes for tighter integration between the framework and tool.

The CLI tool is self-contained in the bin.rs file.

5.1. Commands

The tool works with subcommands, each apt at one very specific task:

- **start** command

  ```
  cronframe start
  ```

  - Start the web server and job scheduler in the background.

- **run** command

  ```
  cronframe run
  ```

  - Run the web server and job scheduler in the terminal.

- **add** command

  ```
  cronframe add '0 0/30 * * * *' 0 my_cli_job.rs
  ```

  - Adds a new CLI job to a CronFrame instance.
  - The arguments it takes are: expression, timeout, job
  - The job is added to the CLI tool instance of cronframe by default.
  - The –port option can be used to target another cronframe instance.

- **load** command

  ```
  cronframe load --file my_job_list.txt
  ```

  - Load jobs from a job definition file.
  - It will perform the same functionality as the add command for each line of the job definition file.
  - If the –file option is not present it will load the job_list.txt in the .cronframe directory.

- **scheduler** command

  ```
  cronframe scheduler start
  ```

  - Perform actions on the scheduler like start and stop
  - It acts by default on the cronframe instance of the CLI tool
  - The –port option can be used to change the target of the command.
- **shutdown** command

  ```
  cronframe shutdown
  ```

  - It shuts down the cronframe instance of the CLI tool.
- **help** command

  ```
  cronframe help scheduler
  ```

  - Print the help message for the tool or the help of the given subcommand

5.2. Tool Information

By default, the CLI tool instance of cronframe will start at http://localhost:8098.

Configuration is available via cronframe.toml inside the .cronframe directory.

The tool itself is written with extensibility in mind.

Each command is self-contained in a function and can take 0 or more arguments upon invocation, this allows for a quick swap of functions to try different versions of the same command should the need arise for it. The modular nature of the commands also allows for splitting the source code of the tool across multiple files should a future version need it because of additional commands or increased complexity of the current ones.

Argument parsing is implemented with the clap library and can be vastly customized to support more subcommands, required arguments, and available options. In addition to keeping things tidy, clap autogenerates a helper utility command that provides the users with information for each registered command as well as their subcommands. A version flag option, -V, is also provided automatically by clap to show the current version of the tool.

OS support for the CLI tool has been tested on both Linux, Ubuntu 22.04 specifically, and on Windows 11. The Rust standard library has a Path type that accounts for the different syntax used by Unix systems and Windows for defining directories in a path.

Much of the work veered on localizing commands specific to a single OS. This is easily done with native Rust macros that allow us to check whether we are compiling the tool for Windows or Linux during installation and therefore supporting more operating systems in the future would simply be a matter of checking for them and providing the os specific code.

While rustc, Rust's compiler, and cargo, Rust's native package manager, used for the compilation of the CLI jobs are commands common to both operating systems, the commands used to copy the final binary of a CLI job inside the .cronframe/cli_jobs directory, are not.

On Linux we use "cp" for this purpose, while on Windows we use "copy", they essentially do the same thing in the end but their context of execution is different as well as their name.

The functionality of the CLI tool has been manually tested only on Ubuntu and Windows, however, since MacOs is very much rooted in Unix as Linux is, the tool should function without significant problems. Should it not, making it work on MacOs is not expected to be too much work. This has not been tested due to a lack of Apple hardware at the moment of writing.

Although many of the libraries used within cronframe have support for many other operating systems, the targeted Operating Systems for the library are Linux distros and the latest Windows version.

The only requirement for installing the tool and for it to function properly is to have cargo installed which will be installed by default on any machine that has the Rust toolchain.

Although the CLI tool could be installed without cargo on the system, it relies on it for some of its more interesting functionality, such as adding CLI jobs. For this reason, there is no alternative method of installation to the one using the cargo install command.

5.3. CLI Jobs

One feature reserved for the tool is the creation of CLI jobs.

A CLI job simply makes a cron job out of Rust source which can be either a single .rs file or an entire project that needs to be built with cargo.

What this essentially means is that our job is whatever is contained inside the main function.

We have two ways of creating a CLI job.

Use the add command of the tool to add a single CLI job:

```
// this uses rustc for compilation
cronframe add '0 0/30 * * * *' 0 path/to/my_cli_job.rs
```

```
// this uses cargo for compilation
cronframe add '0 0/30 * * * *' 0 path/to/my_cli_job
```

Should we have multiple jobs to add, we can save their definition inside a .txt file and load it with the load command.

```
cronframe load --file path/to/my_job_list.txt
```

If the load command does not have the –file option, it will look for a file named “jobs_list.txt” under the .cronframe directory.

6. Documentation and Tutorial

This chapter is mainly focused on tutorials for both the framework and the CLI tool. A link to the official documentation of cronframe is provided, the documentation is not reported here due to it being too cumbersome.

The Documentation for the framework is automatically generated by the Rust doc tool and can be found at https://docs.rs/cronframe, which will show the latest version as well as list all available versions.

The rest of this chapter is set as a tutorial for the use of the framework and the CLI tool on Linux. It will show how to set up the framework in a newly created Rust project and define a few different types of jobs to show what can be done with it.

It will also explain how to run the examples available on the GitHub repo.

For setting up Rust on your machine refer to the official Rust website at https://www.rust-lang.org/tools/install.

The Rust compiler version used for this tutorial is the 1.80.1.

As of writing, the latest version of Cronframe is the 0.1.3.

While not referenced here Windows is also supported and using cronframe on Windows would be nearly identical to using it on Linux.

6.1. Framework Tutorial

6.1.1. Setting Up CronFrame

There are two ways we can set up cronframe:
- using cargo to add it automatically to the project by pulling it from crates.io
- using git to link or clone the repo inside the project

The setup from crates.io, Rust's official crates registry guarantees the use of a working release version of the framework while setting up from git gives the latest updates but no working guarantees as the code might be untested unless the code is the one from a release commit.

6.1.1.1. Set up from Crates.io

The first thing to do is to create a new Rust project with cargo as one normally does with any Rust project.

```
$ cargo new hello_cronframe
```

This will generate a binary crate in the hello_cronframe directory in the current working directory with all the starter code inside it.

To proceed we enter the directory and add the framework:

```
$ cargo add cronframe
```

The last step is to add a crate on which cronframe depends on:

```
$ cargo add linkme@0.3.26
```

Note that the linkme crate has a specific version to be used. Refer to the readme on the GitHub repo for which version of linkme is required or whether it is no longer needed.

6.1.1.2. Set up from GitHub Repo

The setup from the GitHub repo is barely more complicated but requires some care. If you use the latest commit from the master branch then be aware of it possibly not working due to untested code or development of new features.

The first thing to do is to create a new Rust project with cargo as one normally does:

```
$ cargo new hello_cronframe
```

This will generate a binary crate in the hello_cronframe directory in the current working directory with all the starter code inside it.

To proceed we enter the directory and inside the Cargo.toml file we add:

```
#[dependencies]
cronframe = { git = "https://github.com/antcim/cronframe.git"}
```

Again, the last step is adding a crate cronframe depends on:

```
$ cargo add linkme@0.3.26
```

If you want to use the framework from the repo it is recommended to either download the source code from the releases page or clone it from the commit of a release and put it inside your project directory.

To proceed, independently of which way we used to get cronframe, we enter the main directory of our project and inside the Cargo.toml file we add:

```
#[dependencies]
cronframe = { path = "./cronframe", version = "0.1.3"}
```

Again, the last step is adding a crate cronframe depends on:

```
$ cargo add linkme@0.3.26
```

6.1.2. Using CronFrame

Once we have the framework set we only need to decide on what job we want to define.

Let us start with a global job that greets the user with a good morning message every morning at 8 am from Monday to Friday only.

What we need to do first is to import the macros into our project by writing the following code on top of our main.rs file:

```rust
#[macro_use]
extern crate cronframe;
```

This will bring the cron, cron_obj, cron_impl, fn_job, and mt_job macros into scope for us to use.

Since we are defining a global job, for the moment we shall only be concerned with using the cron macro.

It is now time to define our job by writing a function and annotating it with the cron macro:

```rust
#[cron(expr="0 0 8 * * Mon-Fri *", timeout="0")]
fn greeting_job(){
  println!("Have a good morning!");
}
```

Now we have defined the job but we need an instance of the framework so that it can be sent to the scheduler and executed when it is supposed to, so we need to bring into scope the Cron-Frame type:

```rust
use cronframe::CronFrame;
```

Then we need to init cronframe and for that, there is the default function which inits cronframe with no filter for the job type allowed and sets up the default rolling logger.

```rust
fn main(){
  let cronframe = CronFrame::default();
}
```

At this point, the instance will be initialized and our global job will have been built and added to the job pool. The web server available at the default address http://localhost:8098 will also be running but the scheduler is yet to run.

We can run the scheduler with either the start_scheduler method or with the run method. The difference is that start_scheduler returns and does not keep the main thread alive while run keeps main alive.

```rust
fn main(){
  let cronframe = CronFrame::default();
  cronframe.run(); // shorthand for
cronframe.start_scheduler().keep_alive();
}
```

At this point both the webserver and scheduler will be running and the job will be executed when the scheduled time comes.

To recap, the full code will look like the following:

```rust
#[macro_use]
extern crate cronframe;
```

```rust
use cronframe::CronFrame;

#[cron(expr="0 0 8 * * Mon-Fri *", timeout="0")]
fn greeting_job(){
  println!("Have a good morning!");
}

fn main(){
  let cronframe = CronFrame::default();
  cronframe.run(); // shorthand for
cronframe.start_scheduler().keep_alive();
}
```

Now, say we don't want our jobs defined in global space but rather we would like to have a sort of container for them. That can be done with structs by turning them into cron objects.

The import will be the same as the previous one:

```rust
#[macro_use]
extern crate cronframe;
use cronframe::CronFrame;
```

We will now make use of the cron_obj and the cron_impl macros to define a cron object from a struct named Greeting:

```rust
#[cron_obj]
struct Greeting;

#[cron_impl]
impl Greeting{
  // our job goes here
}
```

At this point, we have our container as well as where to put our job. All we need to do is copy and paste the previous global job definition inside the implementation block of Greeting and change to cron macro to the fn_job macro:

```rust
#[cron_obj]
struct Greeting;

#[cron_impl]
impl Greeting{
  #[fn_job(expr="0 0 8 * * Mon-Fri *", timeout="0")]
  fn greeting_job(){
    println!("Have a good morning!");
  }
}
```

At this point, our container with a function job is defined and as expected we need the initialization of the framework which is the same as the previous one:

```rust
fn main(){
  let cronframe = CronFrame::default();
}
```

However, the initialization here does no longer automatically collect our job since it is not a global job anymore. We need to pass the job to the cronframe instance which can be done either before or after we start the scheduler.

Just for the sake of it let us do it after we start the scheduler:

```rust
fn main(){
  let cronframe = CronFrame::default();
  cronframe.start_scheduler();
  Greeting::cf_gather_fn(cronframe.clone());
}
```

We use the cf_gather_fn function automatically generated by the macros on the cron object to pass all function jobs defined in the Greeting cron object to the cronframe instance. Since it is a function job it doesn't even require instantiating the struct for it to work.

At this point, the scheduler is already started and we just need to keep the main thread alive:

```rust
fn main(){
  let cronframe = CronFrame::default();
  cronframe.start_scheduler();
  Greeting::cf_gather_fn(cronframe.clone());
  cronframe.keep_alive();
}
```

To recap, the full code will look like the following:

```rust
#[macro_use]
extern crate cronframe;
use cronframe::CronFrame;

#[cron_obj]
struct Greeting;

#[cron_impl]
impl Greeting{
   #[fn_job(expr="0 0 8 * * Mon-Fri *", timeout="0")]
   fn greeting_job(){
     println!("Have a good morning!");
   }
 }

 fn main(){
   let cronframe = CronFrame::default();
   cronframe.start_scheduler();
   Greeting::cf_gather_fn(cronframe.clone());
```

```
  cronframe.keep_alive();
}
```

We have a nice greeting job that greets everybody at 8 am UTC. But what if we have employees working remotely from different time zones? And what if we wanted to customize the greeting message by adding the name of the employee to it?

For that, we start by turning our function job into a method job by doing the following:

- importing the CronFrameExpr type
- changing the fn_job macro to the mt_job macro
- renaming the job to custom_greeting_job
- removing the timeout attribute in the macro
- adding the self parameter to the function definition

```
#[macro_use]
extern crate cronframe;
use cronframe::{CronFrame, CronFrameExpr};

#[cron_obj]
struct Greeting;

#[cron_impl]
impl Greeting{
   #[mt_job(expr="")]
   fn custom_greeting_job(self){
     println!("Hi ..., have a good morning!");
   }
}
```

As you might have noticed, the expression has been set to an empty string.

Why?

Because we need to write the name of the field of type CronFrameExpr we want to use for that job. So now we add that, and since we're at it we also add a String type field for the name of the employee and use it in our message:

```
#[cron_obj]
struct Greeting{
  employee: String,
  my_expr: CronFrameExpr,
};

#[cron_impl]
impl Greeting{
  #[mt_job(expr="my_expr")]
   fn custom_greeting_job(self){
     println!("Hi {}, have a good morning!", self.employee);
   }
}
```

At this point our method job definition is complete but since this is a method job and methods require an instance of a type to be used, we know that after the usual initialisation of the cronframe instance we need to create an instance for Greeting as well.

This can be done with the new_cron_obj function that is automatically created on the Greeting cron object by the macros. The function takes as arguments the fields that are defined inside the Greeting structure in the exact order they are defined.

```rust
fn main(){
  let cronframe = CronFrame::default();

  let greeting_john = Greeting::new_cron_obj(
    "John".into(), CronFrameExpr::new("0", "0", "8*", "*", "*", "Mon-Fri", "*", 0)
  );
}
```

At this point, we have our cronframe instance with the scheduler not running, and we have our Greeting instance so we need to pass the method job to cronframe either before or after we start the scheduler.

This time around we will pass the job before starting the scheduler:

```rust
fn main(){
  let cronframe = CronFrame::default();
  let greeting_john = Greeting::new_cron_obj(
    "John".into(),
    CronFrameExpr::new("0", "0", "8*", "*", "*", "Mon-Fri", "*", 0)
  );
  greeting_john.cf_gather_mt(cronframe.clone());
  cronframe.run();
}
```

At this point the full code would look like this:

```rust
#[macro_use]
extern crate cronframe;
use cronframe::{CronFrame, CronFrameExpr};

#[cron_obj]
struct Greeting{
  employee: String,
  my_expr: CronFrameExpr,
};

#[cron_impl]
impl Greeting{
  #[mt_job(expr = "my_expr")]
    fn custom_greeting_job(self) {
    println!("Hi {}, have a good morning!", self.employee);
    }
}

fn main(){
  let cronframe = CronFrame::default();
  let greeting_john = Greeting::new_cron_obj(
    "John", CronFrameExpr::new("0", "0", "18*", "*", "*", "Mon-Fri", "*",
0)
  );

  let greeting_jane = Greeting::new_cron_obj(
    "Jane", CronFrameExpr::new("0", "0", "16", "*", "*", "Tue-Thu", "*",
0)
  );

  greeting_john.cf_gather_mt(cronframe.clone());
  greeting_jane.cf_gather_mt(cronframe.clone());

  cronframe.run();
}
```

Notice how we added another greeting job for Jane which has a different cron expression.

What if we wanted to also keep our generic greeting function job?

We put it in the same place it was before and then pass it to the cronframe instance but this time just to keep things brief we are going to use the cf_gather method which collects all method and function jobs defined on a cron object.

```
#[macro_use]
extern crate cronframe;
use cronframe::{CronFrame, CronFrameExpr};

#[cron_obj]
struct Greeting {
  employee: String,
  my_expr: CronFrameExpr,
}

#[cron_impl]
impl Greeting {
  #[fn_job(expr = "0 0 8 * * Mon-Fri *", timeout = "0")]
  fn general_greeting_job() {
    println!("Have a good morning!");
  }

  #[mt_job(expr = "my_expr")]
  fn custom_greeting_job(self) {
    println!("Hi {}, have a good morning!", self.employee);
  }
}

fn main() {
```

```rust
  let cronframe = CronFrame::default();

  let mut greeting_john = Greeting::new_cron_obj(
    "John".into(),
    "0 0 18 * * Mon-Fri * 0".into()
  );

  let mut greeting_jane = Greeting::new_cron_obj(
    "Jane".into(),
    CronFrameExpr::from("0 0 16 * * Tue-Thu * 0")
  );

  greeting_john.cf_gather(cronframe.clone());
  greeting_jane.cf_gather(cronframe.clone());

  cronframe.run();
}

```

Notice how the main function is virtually the same as the previous one except for replacing cf_gather_mt with cf_gather. Also notice that since CronFrameExpr implements the From trait, we can directly turn a string into it by either using the function from on the CronFrameExpr or the method into on a string. This is useful in case we don't require to set each field of the cron expression from different sources such as a different variable for each field.

Due to function jobs being related to a cron object type rather than one of its instances, calling cf_gather twice will only gather the function jobs and pass them to the cronframe instance on the first call.

This is a design choice with the philosophy behind it being that if there is an instance of that type then most likely one wants the functions jobs defined in it to be running as well but if one doesn't then they can simply call the cf_gather_mt and only method jobs will be passed to cronframe.

6.2. Running Examples

If the example is in a single file like "showcase.rs" use the following command:

```
$ cargo run --example showcase
```

If the example is in its crate like `weather_alert` do the following:

```
$ cd examples/weather_alert
$ cargo run
```

Let us now have a look at one of the available examples that shows the framework in use for what might be a real word use.

6.3. Weather Alert Scenario - Example

Now that we have seen how to use cronframe, let us have a look at a slightly more complex job that might resemble some real use case scenario of the framework in an application.

The job we are looking at is a a weather alert job: given the name of a city, we get weather information by making a call to the openweather API and print messages relative to weather conditions.

The job is scheduled to run every ten minutes between 5 am and 7 am and between 2 pm and 4 pm UTC, only from Monday to Friday.

We might want weather alter jobs for different cities so the best way to approach this would be using a method job where we have a cron object with a city field that is used by the job to make the request to the openweather API.

The imports from the library and the cron object definition will look something like the following:

```rust
#[macro_use] extern crate cronframe;

use cronframe::{CronFrame, CronFrameExpr};
use chrono::Local;

#[cron_obj]
struct WeatherAlert {
  city: String,
  schedule: CronFrameExpr,
 }
```

The cron object is very straightforward, we have a city field of type String where the name of our city will be stored and then we have a filed name schedule which is of type CronFrameExpr since we need to provide the schedule for such a job. We added the import of Local from the chrono crate to print the time during each job execution with the Local timezone of the machine that is running the job.

It is now time to define our job.

```rust
#[cron_impl]
impl WeatherAlert {
  #[mt_job(expr = "schedule")]
  fn weather_alert(self) {
    println!("Weather alert job for {}", self.city);

    let url_coord = format!(
      "http://api.openweathermap.org/geo/1.0/direct?q={}&limit={}
&appid={}",
      self.city, 5, API_KEY
     );

     let resp: serde_json::Value = reqwest::blocking::get(url_coord)
     .unwrap()
     .json()
    .unwrap();

    let latitude = resp[0]["lat"].clone();
    let longitude = resp[0]["lon"].clone();

    println!("latitude = {latitude}");
    println!("longitude = {longitude}");

    let url_weather = format!(
      "https://api.openweathermap.org/data/2.5/weather?lat={}&lon={}&appid={}",
      latitude, longitude, API_KEY
    );

    let resp: serde_json::Value = reqwest::blocking::get(url_weather)
      .unwrap()
      .json()
      .unwrap();

    let weather_id: i32 = resp["weather"][0]["id"]
      .clone()
      .to_string()
      .parse()
      .unwrap();

    println!("weather_id = {weather_id}");

    println!("{}: ", Local::now());
    match weather_id {
      200..=232 => println!(
      "!! Thread Carefully: Thunderstorm in {} !!", self.city),
      500..=531 => println!("! Thread Carefully: Rain in {} !", self.city),
      600..=622 => println!("! Thread Carefully: Snow in {} !", self.city),
      781 => println!("!!! Seek Shelter: Tornado in {} !!!", self.city),
      _ => println!("Nothing to worry about in {}", self.city),
    }
  }
}
```

The code for this is highly dependent on your weather service of choice. For openweather we first need to make an API call to get the coordinates of our city and then use those coordinates to make another call to get the weather data of that city.

Weather conditions are coded within numerical ranges for specific conditions, again this is highly dependent on the service used.

In our case, we alert the user about 4 possible conditions just to make things simple.

These conditions are:
- Thunderstorm
- Rain

- Snow
- Tornado

And just to be nice, we also tell them if there is nothing to worry about the weather so they can be relaxed on their way.

What is left to do is init the framework, create an instance of the cron object with our preferred city, and give the job to the framework instance to run.

```rust
fn main() {
 let cronframe = CronFrame::default();

 let alert_schedule = CronFrameExpr::new(
 "0",
 "0/10",
 "5-6,14-15",
 "*",
 "*",
  "Mon-Fri",
  "*",
  0
  );

  let mut venice = WeatherAlert::new_cron_obj("Venice".into(),
alert_schedule);

  venice.cf_gather(cronframe.clone());

  cronframe.run();
 }
```

The whole code will look like this:

```rust
#[macro_use] extern crate cronframe;

use cronframe::{CronFrame, CronFrameExpr};
use chrono::Local;

#[cron_obj]
struct WeatherAlert {
 city: String,
 schedule: CronFrameExpr,
 }

 #[cron_impl]
 impl WeatherAlert {
  #[mt_job(expr = "schedule")]
  fn weather_alert(self) {
  println!("Weather alert job for {}", self.city);

  let url_coord = format!(
  "http://api.openweathermap.org/geo/1.0/direct?q={}&limit={}&appid={}",
  self.city, 5, API_KEY
  );

  let resp: serde_json::Value = reqwest::blocking::get(url_coord)
  .unwrap()
  .json()
  .unwrap();

  let latitude = resp[0]["lat"].clone();
  let longitude = resp[0]["lon"].clone();

  println!("latitude = {latitude}");
  println!("longitude = {longitude}");

  let url_weather = format!(
  "https://api.openweathermap.org/data/2.5/weather?lat={}&lon={}
 &appid={}",
  latitude, longitude, API_KEY
  );

  let resp: serde_json::Value = reqwest::blocking::get(url_weather)
  .unwrap()
  .json()
  .unwrap();

  let weather_id: i32 = resp["weather"][0]["id"]
  .clone()
  .to_string()
  .parse()
  .unwrap();

  println!("weather_id = {weather_id}");

  println!("{}: ", Local::now());
  match weather_id {
  200..=232 => println!(
  "!! Thread Carefully: Thunderstorm in {} !!", self.city),
  500..=531 => println!("! Thread Carefully: Rain in {} !", self.city),
  600..=622 => println!("! Thread Carefully: Snow in {} !", self.city),
  781 => println!("!!! Seek Shelter: Tornado in {} !!!", self.city),
  _ => println!("Nothing to worry about in {}", self.city),
  }
  }
 }

 fn main() {
  let cronframe = CronFrame::default();

  let alert_schedule = CronFrameExpr::new(
  "0",
  "0/10",
  "5-6,14-15",
  "*",
  "*",
  "Mon-Fri",
  "*",
  0
  );

  let mut venice = WeatherAlert::new_cron_obj("Venice".into(),
 alert_schedule);

  venice.cf_gather(cronframe.clone());

  cronframe.run();
 }
```

6.4. CLI Tool Tutorial

6.4.1. Installing CronFrame

There are two ways we can install cronframe:
- install it from crates.io with cargo
- install it from source with cargo

Once again, the recommended way is to install it using crates.io since it will guarantee a working version.

6.4.1.1. Installing from Crates.io

Open your preferred terminal emulator and run the following command:

```
$ cargo install cronframe
```

This will compile the binary for cronframe and put it inside .cargo/bin directory in the home folder.

6.4.1.2. Installing from GitHub Repo

Installing from the repo is easy as well.

The first thing to do is to get the code, either by cloning the repo or downloading a release, the latter is recommended.

Once we have the code, we open a terminal instance inside it and run the following command:

```
$ cargo install --path .
```

We are essentially using cargo to install from a local source, so the binary ends up in the same place as stated above.

6.4.2. Using CronFrame CLI

The CLI tool uses the clap arguments parser library which allows it to have a very user-friendly helper functionality for learning how commands work.

To learn which commands are available we can simply run either of the following:

```
$ cronframe
```

```
$ cronframe help
```

Both of these will print the same message which is a help message containing a list of available commands with their descriptions which for version 0.1.3 looks like this:

```
A Macro Annotation Cron Job Framework with Web Server and CLI Tool.

Usage: cronframe <COMMAND>

Commands:
 start      Start the CronFrame Webserver and Job Scheduler in the background.
 run        Run the CronFrame Webserver and Job Scheduler in the terminal.
 add        Adds a new cli job to a CronFrame instance.
 load       Load jobs from definition file.
 scheduler  Perform actions on the scheduler like start and stop
 shutdown   Shutdown the CronFrame Webserver and Job Scheduler.
 help       Print this message or the help of the given subcommand(s)

Options:
 -h, --help     Print help
 -V, --version  Print version
```

We have six commands at our disposal, each is documented further if we use the help command followed by the command we want to know more of:

```
$ cronframe help add

Adds a new cli job to a CronFrame instance.

Usage: cronframe add [OPTIONS] [EXPR] [TIMEOUT] [JOB]

Arguments:
 [EXPR]     The Cron Expression to use for job scheduling.
 [TIMEOUT]  The value in ms to use for the timeout.
 [JOB]      The path containing the source code of the job.

Options:
 -p, --port <VALUE>
 -h, --help          Print help
 -V, --version       Print version
```

Learning the tool is something the tool itself takes care of which is nice to avoid having to read some lengthy tutorial or documentation, the following explanation will only further clarify what each command does and why it has been made that way.

6.4.2.1. start command

This command is used to start a cronframe instance on the machine. Such an instance is run in the background therefore it is completely independent from the terminal instance that launched it. By default, it will be running at http://127.0.0.1:8098. The scheduler has already started, but unless we load or add jobs to it, it hasn't got much to do. Under the hood, the start command simply spawns the run command in the background so they are virtually the same. Use start when you want your cronframe instance to keep running even if the terminal closes.

6.4.2.2. run command

This command is used to run a cronframe instance on the machine. Such an instance is run in the terminal that launched it. By default, it will be running at http://127.0.0.1:8098. The scheduler has already started, but unless we load or add jobs to it, it hasn't got much to do. Use run when you want your cronframe instance to quit on terminal closure.

6.4.2.3. add command

The add command allows us to make use of a type of job that is available with the CLI tool, that is CLI jobs. A CLI job is an executable built from a Rust source that can either be a single source file or an entire crate.

Using the command requires a cron expression, a timeout, and the path to the source as in the following example:

```
$ cronframe add "0 0/30 9-10 * * Mon-Wed" 0 path/to/my_project_job
```

Note: the cron expression is taken as a single argument so it must be surrounded by double quotes.

6.4.2.4. load command

The load command reads a txt file containing lines of arguments for multiple add commands to execute automatically. If the –file option is not passed to the command, the used file will be jobs_list.txt located inside the .cronframe directory.

An example txt file would be:

```
"0 0/30 9-10 * * Mon-Wed" 0 path/to/my_project_job
"0 0/30 9,12 * * Thu-Fri" 0 path/to/easy_job.rs
"0/5 0 14 * * *" 50000 path/to/timeout.rs
```

6.4.2.5. scheduler command

This command takes an argument that can have only two values:
- start
- stop

There is no need to use it to start the scheduler after either one of the start or the run commands, they take care of it themselves.

A –port option is available to target other possible running instances.

6.4.2.6. shutdown command

This command will provide a graceful shutdown of the instance. It doesn't take a port argument since it is meant to be used only with the cronframe instance created by the CLI tool. Applications that use cronframe as a framework are supposed to handle shutdown on their own.

7. Testing

This chapter presents the testing suite that comes with the framework which is only available if the framework is downloaded from GitHub. Downloading cronframe from crates.io does not include the testing suite since it is not vital for projects that use it.

Since Rust comes with a useful native framework for unit and integration testing [15], several tests have been written to check the functionality of the framework.

There are three categories of tests, each apt at checking one peculiarity of execution:
- standard tests check that a job is run according to the scheduler and without failures
- timeout tests check that a job is run according to the scheduler and it timeouts when it should
- failure tests check that a job is run according to the scheduler and it fails

There is a total of 9 tests, 3 per category of test and 1 per category of jobs, that is to say, one per global, function, and method jobs.

Code in the documentation is also tested which is a feature Rust provides to ensure example code compiles, however, due to it being example code it resents code that is not significant for testing purposes.

Including documentation code, the total number of tests jumps up to 17 total.

The duration of tests has different timespans from tens of seconds to tens of minutes and the entire duration of a full testing execution is a bit more than one hour.

Testing can either be done on the full suite of tests or a single test at a time, in case it is done on the full suite it must be done sequentially with 1 thread of execution since it relies on the logger output and the logger is shared in the testing environment.

Running the entire suite of tests requires downloading the source code from the repo and then running the following command

```
$ cargo test -- --test-threads=1
```

To show extensive test output run:

```
$ cargo test -- --test-threads=1 --show-output
```

To run individual tests we need to first select a submodule of the test suite.

There are 3 of them:
- global
- function
- method

Then we select the test we want to run by its name:
- <job_type>_job_<test_type>

Examples:

```
$ cargo test global::global_job_std
$ cargo test function::function_job_fail
```

```
$ cargo test method::method_job_timeout
```

7.1. Available Test Modules

For a better organization of the suit, it has been split into modules that separate the tests per type of jobs they target.

7.1.1. Tests in the Global Module

The tests available in this module are:

- **global_job_std**
  - it runs for a total of 15 seconds
  - the job is scheduled to run every 5 seconds
  - there is no timeout set
  - it shouldn't fail
- **global_job_fail**
  - it runs for a total of 15 seconds
  - the job is scheduled to run every 5 seconds
  - there is no timeout set
  - it should fail
- **global_job_timeout**
  - it runs for a total of 30 seconds
  - the job is scheduled to run every 5 seconds
  - there is a timeout after 15 seconds from the first job execution
  - it shouldn't fail

7.1.2. Tests in the Function Module

The tests available in this module are:

- **function_job_std**
  - it runs for a total of 5 minutes
  - the job is scheduled to run every 1 minute
  - there is no timeout set
  - it shouldn't fail
- **function_job_fail**
  - it runs for a total of 5 minutes
  - the job is scheduled to run every 1 minute
  - there is no timeout set
  - it should fail
- **function_job_timeout**
  - it runs for a total of 5 minutes
  - the job is scheduled to run every 1 minute
  - there is a timeout after 3 minutes from the first job execution
  - it shouldn't fail

7.1.3. Tests in the Method Module

The tests available in this module are:

- **method_job_std**
  - it runs for a total of 15 minutes
  - the job is scheduled to run every 5 minutes
  - there is no timeout set
  - it shouldn't fail
- **method_job_fail**
  - it runs for a total of 15 minutes
  - the job is scheduled to run every 5 minutes
  - there is no timeout set
  - it should fail
- **method_job_timeout**
  - it runs for a total of 20 minutes
  - the job is scheduled to run every 5 minutes
  - there is a timeout after 12 minutes from the first job execution
  - it shouldn't fail

7.2. Testing Peculiarities

Testing of such a framework was revealed to be less straightforward than expected. The first major problem was the log instance that could be initialized only once in the testing environment. While tests have a separate environment when they execute, the log instance they use is global and shared.

It was necessary to write a function that allowed to init the log only once independently of which test runs first. Since the tests make assertions based on log file timestamps and messages, the log init also had to take care of defining a specific output file for each test by modifying the log configuration.

Rust testing allows for multiple threads of execution to run a testing suite in the least amount of time possible, that is if your tests allow for it. Due to the use of the shared instance of the logger and the tests relying on the output files the logger produces, it is not possible as of the moment of writing to run more than one test at the same time.

Should future logger architecture changes make parallel tests possible, there is the need to account for one more thing. That is having different ports for every running test, otherwise, the initialization of the framework will fail due to a port bind error and only one test will be able to run to completion.

The testing suite could always add more tests, nothing can be tested enough, no matter how extensively tested it is. For this reason, it would perhaps be neat to have some os-dependent tests that target Linux, Windows, and macOS specifically. However since I do not own a Mac, I opted for an os-independent approach to testing which is why there aren't any tests written for the CLI tool. Despite this, I have tested the tool on both Linux and Windows on my machine and the functionality was working as expected.

8. Project Reception

This chapter explains what was behind the publishing process of the framework and CLI tool as well as lists all sites related to cronframe, such as the code repository, the website, and community sharing posts.

The entire code base for the project is available on GitHub at https://github.com/antcim/cronframe under either one of the MIT or Apache 2.0 licenses.

Most Rust libraries, frameworks, and binaries end up on crates.io, and CronFrame followed suit.

The published crates are:
- cronframe
  - this is where both the framework and the CLI tool are contained
- cronframe_macro
  - this is where the macros are contained

The reason for having macros inside their crate is owed to restrictions that Rust currently imposes on crates containing macros which might be lifted in the future.

Ideally, the macro crate will not need to stand on its own and it will be turned into a module of the cronframe crate which is exactly what it was before publishing on crates.io.

By publishing on crates.io, both the framework and CLI tool become available via cargo which is quite convenient for quick and easy installation of either one or both.

The homepage of crates.io also provides visibility for the brief period that the published crates remain in the list of the latest publications or the list of the recently updated ones.

This visibility led to a total number of downloads across four versions of cronframe to be over 1000, in particular, the peak of downloads in a single day is 118 for version 0.1.3.

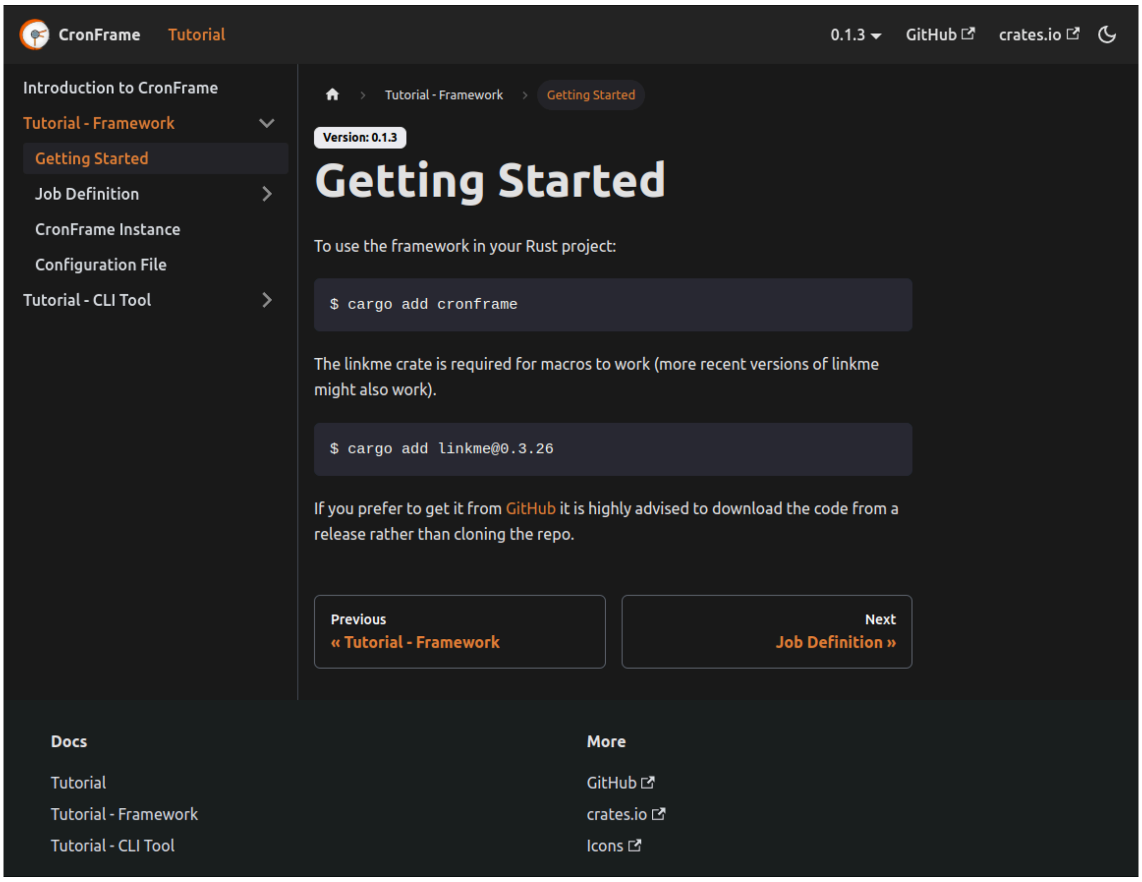
As expected, the greatest number of downloads are observed upon publication of the crate itself or when a new version is published.

To further present the project to Rust developers a post on Rust's subreddit has been posted, available at: https://www.reddit.com/r/rust/comments/1ekjgh6

It received, at the time of writing:
- 15 thousand views
- an upvote of 84%
- 6 shares
- 2 comments

The two comments were illuminating on how the project was perceived by the public and one particularly offered interesting suggestions for possible improvements.

Due to the framework being in early stages the reception it had is more than expected and overall positive since Rust developers are known to be a very opinionated bunch, myself being one of them.



CronFrame Tutorial 0.1.3 GitHub crates.io

Introduction to CronFrame
Tutorial - Framework
Getting Started
Job Definition
CronFrame Instance
Configuration File
Tutorial - CLI Tool

Tutorial - Framework › Getting Started

Version: 0.1.3

Getting Started

To use the framework in your Rust project:

```
$ cargo add cronframe
```

The linkme crate is required for macros to work (more recent versions of linkme might also work).

```
$ cargo add linkme@0.3.26
```

If you prefer to get it from GitHub it is highly advised to download the code from a release rather than cloning the repo.

Previous « Tutorial - Framework
Next Job Definition »

Docs
Tutorial
Tutorial - Framework
Tutorial - CLI Tool

More
GitHub
crates.io
Icons

Figure 4: Tutorial Page of the Website

A website providing a step-by-step tutorial has been made using docusaurus [16] and published with GitHub pages, available at https://antcim.github.io/cronframe_site/, it allows for a more direct dive-in into the workings of the framework and most importantly it supports automatic versioning of the tutorials.

9. Development Summary

This chapter is left as a footnote to any reader to give insight into the thought process that brought about this work of thesis and also delve into new possible additions that future versions of the framework and CLI tool might see.

9.1. Notes

Development started wanting a library that made it possible to use annotation-like syntax on top of functions to define cron jobs but the library base soon evolved into a compact framework quite naturally.

Two main items were the bulk of the work on the framework:
- the annotation macros
- the architecture

Macro syntax has been streamlined as much as possible from early versions that required much more boilerplate code from the user.

The idea for the framework was to make it nearly a plug-and-play dependency which has been achieved as far as:
- small setup for the framework instance
- little code for job definition
- quick configuration

From an architectural perspective, a mixed composition of structs and enums was found to be the better choice, especially for the maintainability of the code in the long run compared to making it full structs or full enums.

No new traits have been defined but Drop and From traits have been implemented for some types for a better integration with the Rust language.

While a cron object's functionality could be thought of as a trait its implementation requires modification of the original struct type which can only be achieved with macros. Injecting all the additional functionality with the macros without a "middleman" trait felt like the right way to do it.

Perhaps a "CronObj" trait would be a neat way to organize what a cron object must contain and have a quick and easy overview of the functionality rather than skimming through macro code but this is much more of a framework developer matter, and most definitely something a user of the framework would overlook.

Supporting only UTC for the job scheduler is a matter of keeping the code more streamlined, it especially makes macros feel as simple as possible both to write and to use.

9.2. Further Development

Further development of the framework could focus on two things:
- new functionalities
- better user experience

As to new functionalities what could be done is a lot, starting from supporting different time-zones for scheduling to the recovery of failed jobs either due to app failure or machine failure.

Custom notifications for job completion to be sent via email or other means might also be a feature of interest but the way they would impact macro definitions might be too cumbersome.

Should the framework evolve to offer quite a lot more, it would also be better to turn some functionalities into opt-in features to be enabled in Cargo.toml. This would allow a user to get only the necessary features for their needs.

Last but not least it would be nice to have a function-like macro for cron objects so that they could be defined as follows:

```
cron_obj!(
 struct User{
 name: String,
 expr: CronFrameExpr,
 }

 #[job(expr="* * * * * * *", timeout="0")]
 fn function_job(){
 // do something...
  }

  #[job(expr="expr")]
  fn function_job(self){
  // do something...
  }
 );
```

This would make a cron object even more immediate to define as well as make the code feel more of a cohesive entity. From a functional standpoint, however, it would simply serve as syntactic sugar to the current definition with the cron_obj and the cron_impl macros. Still, something that might be worth trying.

Some kind of custom syntax to further simplify macros might also be experimented with, though it being custom might be negatively received by any user who strictly wants to use the native syntax of Rust. For this reason, using something like the function macro above would be the safest option of the two.

Bibliography

[1] "Cron CLI." [Online]. Available: https://en.wikipedia.org/wiki/Cron

[2] "Learn Rust." [Online]. Available: https://doc.rust-lang.org/book/

[3] "Cargo Package Manager." [Online]. Available: https://doc.rust-lang.org/cargo/

[4] "Rust Macro System." [Online]. Available: https://doc.rust-lang.org/book/ch19-06-macros.html

[5] "Programming Languages Software Award 2024." [Online]. Available: https://www.sigplan.org/Awards/Software/#2024_The_Rust_Programming_Language

[6] "Developer Survey 2024." [Online]. Available: https://survey.stackoverflow.co/2024/

[7] "Declarative Macros in Rust." [Online]. Available: https://veykril.github.io/tlborm/decl-macros.html

[8] "Procedural Macros in Rust." [Online]. Available: https://veykril.github.io/tlborm/proc-macros.html

[9] "Syn: Rust Parsing Library." [Online]. Available: https://docs.rs/syn

[10] "Quote: Rust Quasi-quoting Library." [Online]. Available: https://docs.rs/quote

[11] "Inventory Crate." [Online]. Available: https://docs.rs/inventory

[12] "Linkme Crate." [Online]. Available: https://docs.rs/linkme

[13] "Rocket Web Framework." [Online]. Available: https://api.rocket.rs/v0.5/rocket/

[14] "Logging Library: log4rs." [Online]. Available: https://docs.rs/log4rs

[15] "Rust Testing." [Online]. Available: https://doc.rust-lang.org/rust-by-example/testing.html

[16] "Docusaurus." [Online]. Available: https://docusaurus.io/docs

Appendix

A - Code Listing for cronframe crate

A-1 - src/lib.rs

```rust
#[doc(hidden)]
#[macro_use] extern crate rocket;

pub use cronframe_macro::{cron, cron_impl, cron_obj, fn_job, mt_job};
#[doc(hidden)]
pub use linkme::distributed_slice;
#[doc(hidden)]
pub use std::{
    any::{self, Any, TypeId},
    sync::{Arc, Mutex},
};

#[doc(hidden)]
pub use crossbeam_channel::{bounded, unbounded, Receiver, Sender};

#[doc(hidden)]
pub use log::info;
#[doc(hidden)]
pub use once_cell::sync::Lazy;
#[doc(hidden)]
pub use std::sync::Once;

// lib modules
pub mod config;
pub mod cronframe;
pub mod cronframe_expr;
pub mod cronjob;
pub mod job_builder;
pub mod logger;
pub mod utils;
pub mod web_server;

// re-export of types
pub use cronframe::CronFrame;
pub use job_builder::JobBuilder;
pub use cronjob::CronJob;
pub use cronframe_expr::CronFrameExpr;

#[doc(hidden)]
pub use inventory::{collect, submit};

// necessary to gather all the global jobs automatically
collect!(JobBuilder<'static>);

/// Used in the init function of the CronJob type to account for the type of job
#[derive(Debug, Clone)]
```

```rust
pub enum CronJobType {
    Global(fn()),
    Method(fn(arg: Arc<Box<dyn Any + Send + Sync>>)),
    Function(fn()),
    CLI
}

/// Used in the init function of the CronFrame type to filter in a single type of job for execution.
///
/// #[macro_use] extern crate cronframe_macro;
/// use cronframe::{CronFilter, CronFrame};
///
/// fn main(){
///     // allow execution of Global Jobs Only
///     let cronframe = CronFrame::init(Some(CronFilter::Global), true);
///     // no filters for the job type
///     //let cronframe = CronFrame::init(None, true);
/// }
///
#[derive(PartialEq, Clone, Copy)]
pub enum CronFilter {
    Global,
    Function,
    Method,
    CLI
}
```

A-2 - src/job_builder.rs

```rust
//! Builder Type for CronJob

use crate::cronjob::CronJob;
use crate::CronJobType;
use chrono::Duration;
use cron::Schedule;
use std::any::Any;
use std::str::FromStr;
use std::sync::Arc;
use uuid::Uuid;

/// Type returned by the helper functions generated by cron, fn_job and mt_job macros.
///
/// It builds a CronJob instance by calling the build method.
#[derive(Debug, Clone)]
pub enum JobBuilder<'a> {
    Global {
        name: &'a str,
        job: fn(),
        cron_expr: &'a str,
        timeout: &'a str,
    },
    Method {
        name: &'a str,
        job: fn(arg: Arc<Box<dyn Any + Send + Sync>>),
        cron_expr: String,
        timeout: String,
        instance: Arc<Box<dyn Any + Send + Sync>>,
    },
    Function {
        name: &'a str,
        job: fn(),
        cron_expr: &'a str,
        timeout: &'a str,
    },

    CLI {
        name: &'a str,
        cron_expr: &'a str,
        timeout: &'a str,
    },
}

impl<'a> JobBuilder<'a> {
    pub const fn global_job(
        name: &'a str,
        job: fn(),
        cron_expr: &'a str,
        timeout: &'a str,
    ) -> Self {
        JobBuilder::Global {
            name,
            job,
            cron_expr,
            timeout,
        }
    }

    pub const fn method_job(
        name: &'a str,
        job: fn(arg: Arc<Box<dyn Any + Send + Sync>>),
        cron_expr: String,
        timeout: String,
        instance: Arc<Box<dyn Any + Send + Sync>>,
    ) -> Self {
        JobBuilder::Method {
            name,
            job,
            cron_expr,
            timeout,
            instance,
        }
    }

    pub const fn function_job(
        name: &'a str,
        job: fn(),
        cron_expr: &'a str,
        timeout: &'a str,
    ) -> Self {
        JobBuilder::Function {
            name,
            job,
            cron_expr,
            timeout,
        }
    }

    pub const fn cli_job(name: &'a str, cron_expr: &'a str, timeout: &'a str) -> Self {
        JobBuilder::CLI {
            name,
            cron_expr,
            timeout,
        }
    }

    // it matches on the job variant to build and builds it
    pub fn build(self) -> CronJob {
        match self {
            Self::Global {
                name,
                job,
                cron_expr,
                timeout,
            } => {
                let schedule =
                    Schedule::from_str(cron_expr)
                      .expect("Failed to parse cron expression!");
                let timeout: i64 = timeout.parse()
                  .expect("Failed to parse timeout!");
                let timeout = if timeout > 0 {
                    Some(Duration::milliseconds(timeout))
                } else {
                    None
                };

                CronJob {
                    name: name.to_string(),
                    id: Uuid::new_v4(),
                    job: CronJobType::Global(job),
                    schedule,
                    timeout,
                    timeout_notified: false,
                    status_channels: Some(crossbeam_channel::bounded(1)),
                    life_channels: None,
                    start_time: None,
                    run_id: None,
                    method_instance: None,
                    failed: false,
                    suspended: false,
                }
            }
            Self::Method {
                name,
                job,
                cron_expr,
                timeout,
                instance,
            } => {
                let schedule =
                    Schedule::from_str(&cron_expr)
                      .expect("Failed to parse cron expression!");
                let timeout: i64 = timeout.parse()
                  .expect("Failed to parse timeout!");
                let timeout = if timeout > 0 {
                    Some(Duration::milliseconds(timeout))
                } else {
                    None
                };

                CronJob {
                    name: name.to_string(),
                    id: Uuid::new_v4(),
                    job: CronJobType::Method(job),
                    schedule,
                    timeout,
                    timeout_notified: false,
                    status_channels:
Some(crossbeam_channel::bounded(1)),
                    life_channels: None,
                    start_time: None,
                    run_id: None,
                    method_instance: Some(instance),
                    failed: false,
                    suspended: false,
                }
            }
            Self::Function {
                name,
                job,
                cron_expr,
                timeout,
            } => {
                let schedule =
                    Schedule::from_str(cron_expr)
                      .expect("Failed to parse cron expression!");
                let timeout: i64 = timeout.parse()
                  .expect("Failed to parse timeout!");
```

```rust
                    let timeout = if timeout > 0 {
                        Some(Duration::milliseconds(timeout))
                    } else {
                        None
                    };

                    CronJob {
                        name: name.to_string(),
                        id: Uuid::new_v4(),
                        job: CronJobType::Function(job),
                        schedule,
                        timeout,
                        timeout_notified: false,
                        status_channels: Some(crossbeam_channel::bounded(1)),
                        life_channels: None,
                        start_time: None,
                        run_id: None,
                        method_instance: None,
                        failed: false,
                        suspended: false,
                    }
                }
                Self::CLI {
                    name,
                    cron_expr,
                    timeout,
                } => {
                    let cron_expr = cron_expr.replace("slh", "/")
                      .replace("%20", " ");
                    let schedule =
                        Schedule::from_str(&cron_expr)
                          .expect("Failed to parse cron expression!");
                    let timeout: i64 = timeout.parse()
                      .expect("Failed to parse timeout!");
                    let timeout = if timeout > 0 {
                        Some(Duration::milliseconds(timeout))
                    } else {
                        None
                    };

                    CronJob {
                        name: name.to_string(),
                        id: Uuid::new_v4(),
                        job: CronJobType::CLI,
                        schedule,
                        timeout,
                        timeout_notified: false,
                        status_channels: Some(crossbeam_channel::bounded(1)),
                        life_channels: None,
                        start_time: None,
                        run_id: None,
                        method_instance: None,
                        failed: false,
                        suspended: false,
                    }
                }
            }
        }
    }

```

A-3 - src/cronjob.rs

```rust
//! CronJob type, built by JobBuilder

use crate::{utils, CronJobType};
use chrono::{DateTime, Duration, Utc};
use cron::Schedule;
use crossbeam_channel::{Receiver, Sender};
use std::{any::Any, process::Command,
  str::FromStr, sync::Arc, thread::JoinHandle};
use uuid::Uuid;

/// This type collects all necessary data
/// for a cron job to be used in the scheduler.
///
/// While it could be used directly there are macros that build jobs for you.
#[derive(Debug, Clone)]
pub struct CronJob {
    pub suspended: bool,
    pub name: String,
    pub id: Uuid,
    pub job: CronJobType,
    pub schedule: Schedule,
    pub timeout: Option<Duration>,
    pub timeout_notified: bool,
    pub status_channels: Option<(Sender<String>, Receiver<String>)>,
    pub life_channels: Option<(Sender<String>, Receiver<String>)>,
    pub start_time: Option<DateTime<Utc>>,
    pub run_id: Option<Uuid>,
    pub method_instance: Option<Arc<Box<dyn Any + Send + Sync>>>,
    pub failed: bool,
}

impl CronJob {
    // this function is used in the scheduler thread to get a handle if the job has to be scheduled
    pub fn try_schedule(&mut self, grace_period: u32) -> Option<JoinHandle<()>> {
        if self.check_schedule() {
            self.run_id = Some(Uuid::new_v4());
            //self.status_channels = Some(crossbeam_channel::bounded(1));

            if self.start_time.is_none() {
                self.start_time = Some(Utc::now());
            }

            // we try to schedule the job and return its handle
            // in case scheduling fails for any conflict
            // we try again for as long as we are in the gracefull period

            if let Ok(handle) = self.run() {
                return Some(handle);
            }

            let gracefull_period = grace_period as i64;
            let first_try = Utc::now();
            let limit_time = first_try + Duration::milliseconds(gracefull_period);
            let mut graceful_log = false;

            while Utc::now() < limit_time {
                match self.run_graceful() {
                    Ok(handle) => {
                        if graceful_log {
                            let job_id = format!("{} ID#{}", self.name, self.id);
                            let run_id = self
                                .run_id
                                .expect("run_id unwrap error in try_schedule")
                                .to_string();
                            info!("job @{job_id} RUN_ID#{run_id} - Scheduled in Graceful Period");
                        }
                        return Some(handle);
                    }
                    Err(_error) => graceful_log = true,
                }
            }
        }
        None
    }

    // checks if a job's upcoming schedule is within the next second
    pub fn check_schedule(&self) -> bool {
        let now = Utc::now();
        if let Some(next) = self.schedule.upcoming(Utc).take(1).next() {
            let until_next = (next - now).num_milliseconds();
            if until_next <= 1000 {
                return true;
            }
        }
        false
    }

    // the expected value is in milliseconds
    pub fn set_timeout(&mut self, value: i64) {
        self.timeout = if value > 0 {
            Some(Duration::milliseconds(value))
        } else {
            None
        };
    }

    // used to retrive a job's status to display it in the web server
    pub fn status(&self) -> String {
        if self.suspended {
            "Suspended".to_string()
        } else if self.check_timeout() {
            "Timed-Out".to_string()
        } else if self.run_id.is_some() {
            "Running".to_string()
        } else {
            "Awaiting Schedule".to_string()
        }
    }

    // method used by the web server the change the cron expression of a job
    pub fn set_schedule(&mut self, expression: &str) -> bool {
        let expr = expression.replace("slh", "/").replace("%20", " ");
        if let Ok(schedule) = Schedule::from_str(expr.as_str()) {
            self.schedule = schedule;
            return true;
        }
        false
    }

    // returns true if timeout expired
    pub fn check_timeout(&self) -> bool {
        if let Some(timeout) = self.timeout {
            if self.start_time.is_some() {
                let timeout = self
                    .start_time
                    .expect("start time unwrap error in check_timeout")
                    + timeout;

                if Utc::now() >= timeout {
                    return true;
                }
            }
        }
        false
    }

    // it resets the timeout if 24h have passed
    pub fn reset_timeout(&mut self) {
        if let Some(timeout) = self.timeout {
            if self.start_time.is_some() {
                let timeout = self
                    .start_time
                    .expect("start time unwrap error in timeout_reset")
                    + timeout;

                if Utc::now() >= timeout + Duration::hours(24) {
                    self.start_time = None;
                }
            }
        }
    }

    // get the schedule constructed from the cron expression
    pub fn schedule(&self) -> String {
        self.schedule.to_string()
    }

    // it returns the timeout or "None" if a timeout is not set
    pub fn timeout_to_string(&self) -> String {
        if self.timeout.is_some() {
            let timeout = self
                .timeout
                .expect("timeout unwrap error in timeout_to_string");
            format!(
                "{} s \n {} ms",
                timeout.num_seconds(),
                timeout.num_milliseconds()
            )
        } else {
            "None".into()
        }
    }

    // spells out the type of the job
    pub fn type_to_string(&self) -> String {
        match self.job {
            CronJobType::Global(_) => "Global".to_string(),
            CronJobType::Function(_) => "Function".to_string(),
            CronJobType::Method(_) => "Method".to_string(),
            CronJobType::CLI => "CLI".to_string(),
        }
    }

    // if the job is active it returns the schedule otherwise a message
telling why there is no next schedule
    pub fn upcoming_utc(&self) -> String {
        if self.suspended {
            return "None due to scheduling suspension.".to_string();
        } else if self.check_timeout() {
            return "None due to timeout.".to_string();
        }
```

```rust
        self.schedule
            .upcoming(Utc)
            .into_iter()
            .next()
            .expect("schedule unwrap error in upcoming_utc")
            .to_string()
    }

    // if the job is active it returns the schedule otherwise a message telling why there is no next schedule
    pub fn upcoming_local(&self) -> String {
        if self.suspended {
            return "None due to scheduling suspension.".to_string();
        } else if self.check_timeout() {
            return "None due to timeout.".to_string();
        }
        utils::local_time(
            self.schedule
                .upcoming(Utc)
                .into_iter()
                .next()
                .expect("schedule unwrap error in upcoming_utc"),
        )
        .to_string()
    }

    // it returns the id of the current execution of the job, or "None" if it is not running
    pub fn run_id(&self) -> String {
        match &self.run_id {
            Some(uuid) => uuid.to_string(),
            None => "None".into(),
        }
    }

    // this spawns a control thread for the job that spawns a thread with the actual job
    pub fn run(&self) -> std::io::Result<JoinHandle<()>> {
        let cron_job = self.clone();
        let tx = self
            .status_channels
            .as_ref()
            .expect("tx unwap error in job run method")
            .0
            .clone();
        let _rx = self
            .status_channels
            .as_ref()
            .expect("rx unwap error in job run method")
            .1
            .clone();
        let schedule = self.schedule.clone();
        let job_id = format!("{} ID#{}", self.name, self.id);
        let run_id = self
            .run_id
            .as_ref()
            .expect("run_id unwap error in job run method")
            .clone();

        // the actual job thread
        // this is spawned form the control thread
        // it gets the next schedule, waits up to it and runs the job
        let job_thread = move || {
            let now = Utc::now();
            if let Some(next) = schedule.upcoming(Utc).take(1).next() {
                let until_next = next - now;
                std::thread::sleep(until_next.to_std().unwrap());
                info!("job @{job_id} RUN_ID#{run_id} - Execution");
                match cron_job.job {
                    CronJobType::Global(job) | CronJobType::Function(job) => job(),
                    CronJobType::Method(job) => job(cron_job
                        .method_instance
                        .expect("method instance unwrap error in job thread")),
                    CronJobType::CLI => {
                        let home_dir = {
                            let tmp = home::home_dir().unwrap();
                            tmp.to_str().unwrap().to_owned()
                        };
                        let _build = Command::new(format!("./{}", cron_job.name))
                            .current_dir(format!("{home_dir}/.cronframe/cli_jobs"))
                            .status()
                            .expect("process failed to execute");
                    }
                }
            }
        };

        // the control thread handle is what gets returned to the cronframe
        // this allows to check for job completion or fail
        let control_thread = move || {
            let job_handle = std::thread::spawn(job_thread);

            while !job_handle.is_finished() {
                std::thread::sleep(Duration::milliseconds(250).to_std().unwrap());
            }

            match job_handle.join() {
                Ok(_) => {
                    let _ = tx.send("JOB_COMPLETE".to_string());
                }
                Err(_) => {
                    let _ = tx.send("JOB_ABORT".to_string());
                }
            };
        };

        std::thread::Builder::spawn(std::thread::Builder::new(), control_thread)
    }

    // same as run but it accounts for graceful period
    pub fn run_graceful(&self) -> std::io::Result<JoinHandle<()>> {
        let cron_job = self.clone();
        let tx = self
            .status_channels
            .as_ref()
            .expect("tx unwap error in job run_graceful method")
            .0
            .clone();
        let _rx = self
            .status_channels
            .as_ref()
            .expect("rx unwap error in job run_graceful method")
            .1
            .clone();
        let schedule = self.schedule.clone();
        let job_id = format!("{} ID#{}", self.name, self.id);
        let run_id = self.run_id.as_ref().unwrap().clone();

        // the actual job thread in graceful period
        // this is spawned form the control thread
        // it runs right away
        let job_thread = move || {
            let now = Utc::now();
            if let Some(next) = schedule.upcoming(Utc).take(1).next() {
                let until_next = next - now;

                // we sleep only if we haven't yet reached the scheduled time
                // otherwise we immediatly execute the job
                if now < next {
                    std::thread::sleep(until_next.to_std().unwrap());
                }

                info!("job @{job_id} RUN_ID#{run_id} - Execution");
                match cron_job.job {
                    CronJobType::Global(job) | CronJobType::Function(job) => job(),
                    CronJobType::Method(job) => job(cron_job
                        .method_instance
                        .expect("method instance unwrap error in job thread")),
                    CronJobType::CLI => {
```

```rust
                        let home_dir = {
                            let tmp = home::home_dir().unwrap();
                            tmp.to_str().unwrap().to_owned()
                        };
                        let _build = Command::new(format!("./{}", cron_job.name))
                            .current_dir(format!("{home_dir}/.cronframe/cli_jobs"))
                            .status()
                            .expect("process failed to execute");
                    }
                }
            }
        };

        // the control thread handle is what gets returned to the cronframe
        // this allows to check for job completion or fail
        let control_thread = move || {
            let job_handle = std::thread::spawn(job_thread);

            while !job_handle.is_finished() {
                std::thread::sleep(Duration::milliseconds(250).to_std().unwrap());
            }

            match job_handle.join() {
                Ok(_) => {
                    let _ = tx.send("JOB_COMPLETE".to_string());
                }
                Err(_) => {
                    let _ = tx.send("JOB_ABORT".to_string());
                }
            };
        };

        std::thread::Builder::spawn(std::thread::Builder::new(), control_thread)
    }
}

```

A-4 - src/cronframe.rs

```rust
//! The Core Type of the Framework

use crate::{
    config::read_config, cronjob::CronJob, job_builder::JobBuilder, logger, utils, web_server,
    CronFilter, CronJobType,
};
use chrono::Duration;
use crossbeam_channel::{Receiver, Sender};
use rocket::Shutdown;
 use std::{
     collections::HashMap,
     sync::{Arc, Mutex},
     thread::JoinHandle,
 };

 const GRACE_DEFAULT: u32 = 250;

 /// This is the type that provides the scheduling and management of jobs.
 ///
 /// It needs to be initialised once to setup the web server and gather global jobs.
 ///
 /// Either one of the `start_scheduler` or `run` method must be invoked for it to actually start.
 ///
 /// # #[macro_use] extern crate cronframe_macro;
 /// # use cronframe::CronFrame;
 /// fn main(){
 ///     let cronframe = CronFrame::default(); // this a shorthand for Cronframe::init(None, true);
 ///     cronframe.start_scheduler(); //does not keep main alive
 ///     //cronframe.keep_alive(); // keeps main thread alive
 ///     //cronframe.run(); //starts the scheduler, keeps main alive
 /// }
 pub struct CronFrame {
     pub cron_jobs: Mutex<Vec<CronJob>>,
     job_handles: Mutex<HashMap<String, JoinHandle<()>>>,
     _logger: Option<log4rs::Handle>,
     pub web_server_channels: (Sender<Shutdown>, Receiver<Shutdown>),
     pub filter: Option<CronFilter>,
     server_handle: Mutex<Option<Shutdown>>,
     pub quit: Mutex<bool>,
     pub grace: u32,
     pub running: Mutex<bool>,
 }

 impl CronFrame {
     /// It returns an `Arc<CronFrame>` which is used in the webserver and can be used to start the scheduler.
     ///
     /// # #[macro_use] extern crate cronframe_macro;
     /// # use cronframe::CronFrame;
     /// fn main(){
     ///     // inits the framework instance and gathers global jobs if there are any
     ///     // does not start the scheduler, only the web server is live
     ///     let cronframe = CronFrame::default(); // this a shorthand for Cronframe::init(None, true);
     ///     //cronframe.keep_alive(); // keeps main thread alive
     ///     //cronframe.run(); //starts the scheduler, keeps main alive
     /// }
     ///
     pub fn default() -> Arc<CronFrame> {
         CronFrame::init(None, true)
     }

     /// Init function of the framework, it takes two agruments:
     ///
     /// filter: Option<CronFilter>
     /// use_logger: bool
     ///
     ///
     /// It manages:
     /// - the logger setup if use_logger is true
     /// - the creation of the CronFrame Instance
     /// - the collection of global jobs
     /// - the setup of the web server
     ///
     /// It returns an `Arc<CronFrame>` which is used in the webserver and
 to start the scheduler.
     pub fn init(filter: Option<CronFilter>, use_logger: bool) ->
 Arc<CronFrame> {
         println!("Starting CronFrame...");

         let logger = if use_logger {
             Some(logger::rolling_logger())
         } else {
             None
         };

         let frame = CronFrame {
             cron_jobs: Mutex::new(vec![]),
             job_handles: Mutex::new(HashMap::new()),
             _logger: logger,
             web_server_channels: crossbeam_channel::bounded(1),
             filter,
             server_handle: Mutex::new(None),
             quit: Mutex::new(false),
             grace: {
                 if let Some(config_data) = read_config() {
                     if let Some(scheduler_data) = config_data.scheduler {
                         scheduler_data.grace.unwrap_or_else(|| 250)
                     } else {
                         GRACE_DEFAULT
                     }
                 } else {
                     GRACE_DEFAULT
                  }
              },
              running: Mutex::new(false),
          };

          info!("CronFrame Init Start");
```

```rust
        info!("Graceful Period {} ms", frame.grace);
        info!("Colleting Global Jobs");

        for job_builder in inventory::iter::<JobBuilder> {
            let cron_job = job_builder.clone().build();
            info!("Found Global Job \"{}\"", cron_job.name);
            frame
                .cron_jobs
                .lock()
                .expect("global job gathering error during init")
                .push(cron_job)
        }

        info!("Global Jobs Collected");
        info!("CronFrame Init Complete");

        info!("CronFrame Server Init");
        let frame = Arc::new(frame);
        let server_frame = frame.clone();

        let running = Mutex::new(false);

        std::thread::spawn(move || web_server::web_server(server_frame));

        *frame
            .server_handle
            .lock()
            .expect("web server handle unwrap error") = match frame.web_server_channels.1.recv() {
            Ok(handle) => {
                *running.lock().unwrap() = true;
                Some(handle)
            }
            Err(error) => {
                error!("Web server shutdown handle error: {error}");
                None
            }
        };

        if *running.lock().unwrap() {
            let (ip_address, port) = utils::ip_and_port();
            info!(
                "CronFrame Web Server running at http://{}:{}",
                ip_address, port
            );
            println!("CronFrame running at http://{}:{}", ip_address, port);
        }

        frame
    }

    /// It adds a CronJob instance to the job pool
    /// Used in the cf_gather_mt and cf_gather_fn
    pub fn add_job(self: &Arc<CronFrame>, job: CronJob) -> Arc<CronFrame> {
        self.cron_jobs
            .lock()
            .expect("add_job unwrap error on lock")
            .push(job);
        self.clone()
    }

    // It crates a new job classified as a global job and adds it to the job pool
    pub fn new_job(
        self: Arc<CronFrame>,
        name: &str,
        job: fn(),
        cron_expr: &str,
        timeout: &str,
    ) -> Arc<CronFrame> {
        self.add_job(JobBuilder::global_job(name, job, cron_expr, timeout).build())
    }

    /// It spawns a thread that manages the scheduling of the jobs and termination of jobs.
    ///
    /// This method returns after spawning the scheduler.
    ///
    /// Keeping the main thread alive is left to the user.
    ///
    /// Use the `run` method to spawn the scheduler and keep the main thread alive.
    ///
    /// # #[macro_use] extern crate cronframe_macro;
    /// # use cronframe::CronFrame;
    /// fn main(){
    ///     let cronframe = CronFrame::default().start_scheduler();
    /// }
    ///
    pub fn start_scheduler<'a>(self: &Arc<Self>) -> Arc<Self> {
        let cronframe = self.clone();
        let ret = cronframe.clone();

        // if already running, return
        if *self.running.lock().unwrap() {
            return ret;
        }

        *cronframe
            .running
            .lock()
            .expect("running unwrap error in quit start_scheduler
method") = true;
        *cronframe
            .quit
            .lock()
            .expect("quit unwrap error in start_scheduler method") = false;

        let scheduler = move || loop {
            // sleep some otherwise the cpu consumption goes to the moon
            std::thread::sleep(Duration::milliseconds(500).to_std().unwrap());

            if *cronframe
                .quit
                .lock()
                .expect("quit unwrap error in scheduler")
            {
                break;
            }

            if !*cronframe
                .running
                .lock()
                .expect("quit unwrap error in scheduler")
            {
                break;
            }

            let mut cron_jobs = cronframe
                .cron_jobs
                .lock()
                .expect("cron jobs unwrap error in scheduler");
            let mut jobs_to_remove: Vec<usize> = Vec::new();

            for (i, cron_job) in &mut (*cron_jobs).iter_mut().enumerate() {
                if let Some(filter) = &cronframe.filter {
                    let job_type = match cron_job.job {
                        CronJobType::Global(_) => CronFilter::Global,
                        CronJobType::Function(_) => CronFilter::Function,
                        CronJobType::Method(_) => CronFilter::Method,
                        CronJobType::CLI => CronFilter::CLI,
                    };

                    if job_type != *filter {
                        continue;
                    }
                }

                let job_id = format!("{} ID#{}", cron_job.name, cron_job.id);

```

```rust
                    // if cron_obj instance related to the job is dropped delete the job
                    let to_be_deleted = if let Some((_, life_rx)) = cron_job.life_channels.clone() {
                        match life_rx.try_recv() {
                            Ok(message) => {
                                if message == "JOB_DROP" {
                                    info!("job @{} - Dropped", job_id);
                                    jobs_to_remove.push(i);
                                    true
                                } else {
                                    false
                                }
                            }
                            Err(_error) => false,
                        }
                    } else {
                        false
                    };

                    // if the job_id key is not in the hashmap then attempt to schedule it
                    // if scheduling is a success then add the key to the hashmap

                    let mut job_handlers = cronframe
                        .job_handles
                        .lock()
                        .expect("job handles unwrap error in scheduler");

                    // check if the daily timeout expired and reset it if need be
                    cron_job.reset_timeout();

                    // if there is no handle for the job see if it need to be scheduled
                    if !job_handlers.contains_key(&job_id) && !to_be_deleted {
                        if cron_job.suspended {
                            continue;
                        }

                        // if the job timed-out than skip to the next job
                        if cron_job.check_timeout() {
                            if !cron_job.timeout_notified {
                                info!("job @{} - Reached Timeout", job_id);
                                cron_job.timeout_notified = true;
                            }
                            continue;
                        }

                        let handle = (*cron_job).try_schedule(cronframe.grace);

                        if handle.is_some() {
                            job_handlers.insert(
                                job_id.clone(),
                                handle.expect("job handle unwrap error after try_schedule"),
                            );
                            info!(
                                "job @{} RUN_ID#{} - Scheduled",
                                job_id,
                                cron_job.run_id.as_ref().expect("run_id unwrap error")
                            );
                        }
                    }
                    // the job is in the hashmap and running
                    // check to see if it sent a message that says it finished or aborted
                    else if let Some((_, status_rx)) = cron_job.status_channels.clone() {
                        match status_rx.try_recv() {
                            Ok(message) => {
                                if message == "JOB_COMPLETE" {
                                    info!(
                                        "job @{} RUN_ID#{} - Completed",
                                        job_id,
                                        cron_job.run_id.as_ref().unwrap()
                                    );
                                    job_handlers.remove(job_id.as_str());
                                    cron_job.run_id = None;
                                } else if message == "JOB_ABORT" {
                                    info!(
                                        "job @{} RUN_ID#{} - Aborted",
                                        job_id,
                                        cron_job.run_id.as_ref().unwrap()
                                    );
                                    job_handlers.remove(job_id.as_str());
                                    cron_job.run_id = None;
                                    cron_job.failed = true;
                                }
                            }
                            Err(_error) => {}
                        }
                    }
                }

                // cleanup of dropped method jobs
                if !jobs_to_remove.is_empty() {
                    let num_jobs = jobs_to_remove.len();
                    for i in 0..num_jobs {
                        cron_jobs.remove(jobs_to_remove[i]);
                        for j in i + 1..num_jobs {
                            jobs_to_remove[j] -= 1;
                        }
                    }
                }
            };

            std::thread::spawn(scheduler);
            info!("CronFrame Scheduler Running");
            ret
        }

        /// This function can be used to keep the main thread alive after the scheduler has been started
        pub fn keep_alive(self: &Arc<Self>) {
            loop {
                std::thread::sleep(Duration::milliseconds(500).to_std().unwrap());
                if *self.quit.lock().unwrap() {
                    break;
                }
            }
        }

        /// Blocking method that starts the scheduler and keeps the main thread alive
        /// Use the `start_scheduler` method if need to start the scheduler and
        /// retain control of execution in main
        pub fn run(self: &Arc<Self>) {
            self.start_scheduler().keep_alive();
        }

        /// It quits the running scheduler instance
        pub fn stop_scheduler(self: &Arc<Self>) {
            info!("CronFrame Scheduler Shutdown");
            *self.running.lock().unwrap() = false;
        }

        /// Function to call for a graceful shutdown of the framework instance
        ///
        /// # #[macro_use] extern crate cronframe_macro;
        /// # use cronframe::CronFrame;
        ///
        /// fn main(){
        ///     let cronframe = CronFrame::default();
        ///     // do somthing...
        ///     cronframe.start_scheduler();
        ///     // do other things...
        ///     cronframe.quit();
        /// }
        ///
        pub fn quit(self: &Arc<Self>) {
            self.stop_scheduler();
            info!("CronFrame Shutdown");

            // wait for job handlers to finisH
            let cronframe = self.clone();

            let handles = cronframe
                .job_handles
                .lock()
                .expect("job handles unwrap error in stop scheduler 
    method");

            for handle in handles.iter() {
                while !handle.1.is_finished() {
                    // do some waiting until all job threads have 
    terminated.
                }
            }

            // quit the web server
            self.server_handle
                .lock()
                .expect("web server unwrap error in quit method")
                .clone()
                .expect("web server unwrap error after clone in quit 
    method")
                .notify();

            *self
                .quit
                .lock()
                .expect("quit unwrap error in stop scheduler method") = 
    true;
        }
    }

```

A-5 - src/cronframe_expr.rs

```rust
//! This type is used in cron objects to define the cron expression and 
timeout for a method job.

/// It implements the From trait for strings
#[derive(Debug, Clone, Default)]
pub struct CronFrameExpr {
    seconds: String,
    minutes: String,
    hour: String,
    day_month: String,
     month: String,
     day_week: String,
     year: String,
     timeout: u64,
}

impl CronFrameExpr {
    /// Creates a new CronFrameExpr instance where:
    /// - s   is seconds
    /// - m   is minutes
    /// - h   is hour
    /// - dm  is day_month
    /// - mth is month
    /// - dw  is day_week
    /// - y   is year
    /// - t   is timeout
    ///
    ///
    /// use cronframe::CronFrameExpr;
    /// fn main(){
    ///     let my_expr = CronFrameExpr::new("0", "5", "10-14", "*", "*", "Sun", "*", 0);
    /// }
    ///
    pub fn new(s: &str, m: &str, h: &str, dm: &str, mth: &str, dw: &str, y: &str, t: u64) -> Self {
        CronFrameExpr {
            seconds: s.to_string(),
            minutes: m.to_string(),
            hour: h.to_string(),
            day_month: dm.to_string(),
            month: mth.to_string(),
            day_week: dw.to_string(),
            year: y.to_string(),
            timeout: t,
        }
    }

    pub fn expr(&self) -> String {
        format!(
            "{} {} {} {} {} {} {}",
            self.seconds,
            self.minutes,
            self.hour,
            self.day_month,
            self.month,
            self.day_week,
            self.year
        )
    }

    pub fn timeout(&self) -> u64 {
        self.timeout
    }
}

impl From<&str> for CronFrameExpr {
    fn from(item: &str) -> Self {
        let items: Vec<_> = item.split(" ").collect();
        CronFrameExpr::new(
            items[0],
            items[1],
            items[2],
            items[3],
            items[4],
            items[5],
            items[6],
            items[7].parse().unwrap(),
        )
    }
}
```

A-6 - src/web_server.rs

```rust
//! Custom setup of rocket.rs for the Cronframe web server

use crate::{
    config::read_config, cronframe::CronFrame, utils, CronFilter, 
CronJobType, JobBuilder,
};
use colored::Colorize;
use log::info;
use rocket::{
    config::Shutdown,
    figment::{
        providers::{Env, Format, Toml},
        Figment, Profile,
    },
    serde::Serialize,
};
use rocket_dyn_templates::{context, Template};
use std::{fs, path::Path, sync::Arc, time::Duration};

/// Called by the init funciton of the Cronframe type for setting up the 
web server
///
/// It provides 7 routes, five of which are API only.
///
/// Upon first start of the framework it will generate a templates 
directory.
pub fn web_server(frame: Arc<CronFrame>) {
    generate_template_dir();

    let cronframe = frame.clone();

    let tokio_runtime = rocket::tokio::runtime::Runtime::new().unwrap();

    let base_config = match read_config() {
        Some(config_data) => rocket::Config {
            port: {
                if let Some(webserver_data) = &config_data.webserver {
                    webserver_data.port.unwrap_or_else(|| 8098)
                } else {
                    8098
                }
            },
            address: {
                if let Some(webserver_data) = config_data.webserver {
                    webserver_data.ip.unwrap_or_else(|| "127.0.0.1".to_string())
                } else {
                    "127.0.0.1".to_string()
                }
                .parse()
                .unwrap()
            },
            shutdown: Shutdown {
                ctrlc: false,
                ..Default::default()
            },
            cli_colors: false,
            ..rocket::Config::release_default()
        },
        None => {
            // default config
            rocket::Config {
                port: 8098,
                address: std::net::Ipv4Addr::new(127, 0, 0, 1).into(),
                shutdown: Shutdown {
                    ctrlc: false,
                    ..Default::default()
                },
                cli_colors: false,
                ..rocket::Config::release_default()
            }
        }
    };

    let ip_address = base_config.address;
    let port = base_config.port;
    let home_dir = utils::home_dir();

    if std::env::var("CRONFRAME_CLI").is_ok() {
        std::env::set_var(
            "ROCKET_CONFIG",
            format!("{home_dir}/.cronframe/rocket.toml"),
        );
    }

    let config = Figment::from(base_config)
        .merge(Toml::file(Env::var_or("ROCKET_CONFIG", "Rocket.toml")).nested())
        .merge(Env::prefixed("ROCKET_").ignore(&["PROFILE"]).global())
        .select(Profile::from_env_or(
            "ROCKET_PROFILE",
            rocket::Config::DEFAULT_PROFILE,
        ));

    let rocket = rocket::Rocket::custom(config)
        .mount(
            "/",
            routes![
                styles,
                cronframe,
                tingle,
                tinglejs,
                home,
                job_info,
                update_timeout,
                update_schedule,
                suspension_handle,
                start_scheduler,
                stop_scheduler,
                add_cli_job,
                shutdown,
            ],
        )
        .attach(Template::fairing())
        .manage(frame);

    let (tx, _) = cronframe.web_server_channels.clone();

    tokio_runtime.block_on(async move {
        let rocket = rocket.ignite().await.expect("Ignition Error");

        let _ = tx.send(rocket.shutdown());

        match rocket.launch().await {
            Ok(_) => println!("CronFrame running at http://{}:{}",
ip_address, port),
            Err(err) => {
                println!(
                    "{} {}",
                    "Error during CronFrame init:".red().bold(),
                    err.pretty_print()
                );
                cronframe.clone().quit();
            }
        }
    });
}

// necessary to have somewhat decent-looking pages
#[get("/styles")]
async fn styles() -> Result<rocket::fs::NamedFile, std::io::Error> {
    if std::env::var("CRONFRAME_CLI").is_ok() {
        let home_dir = utils::home_dir();
        rocket::fs::NamedFile::open(format!("{home_dir}/.cronframe/templates/styles.css")).await
    } else {
        rocket::fs::NamedFile::open(format!("./templates/styles.css")).await
    }
}

// necessary to have somewhat decent-looking pages
#[get("/tingle")]
async fn tingle() -> Result<rocket::fs::NamedFile, std::io::Error> {
    if std::env::var("CRONFRAME_CLI").is_ok() {
        let home_dir = utils::home_dir();
        rocket::fs::NamedFile::open(format!("{home_dir}/.cronframe/templates/tingle.css")).await
    } else {
        rocket::fs::NamedFile::open(format!("./templates/tingle.css")).await
    }
}

// necessary to have decent modals
#[get("/tinglejs")]
async fn tinglejs() -> Result<rocket::fs::NamedFile, std::io::Error> {
    if std::env::var("CRONFRAME_CLI").is_ok() {
        let home_dir = utils::home_dir();
        rocket::fs::NamedFile::open(format!("{home_dir}/.cronframe/templates/tingle.js")).await
    } else {
        rocket::fs::NamedFile::open(format!("./templates/tingle.js")).await
    }
}

// necessary to have somewhat functioning pages
#[get("/cronframejs")]
async fn cronframe() -> Result<rocket::fs::NamedFile, std::io::Error> {
    if std::env::var("CRONFRAME_CLI").is_ok() {
        let home_dir = utils::home_dir();
        rocket::fs::NamedFile::open(format!("{home_dir}/.cronframe/templates/cronframe.js")).await
    } else {
        rocket::fs::NamedFile::open(format!("./templates/cronframe.js")).await
    }
}

#[derive(Serialize)]
#[serde(crate = "rocket::serde")]
struct JobList {
    name: String,
```

```rust
    id: String,
}

// homepage returning a list of al jobs in the following categories: active, timed out, suspended
#[get("/")]
fn home(cronframe: &rocket::State<Arc<CronFrame>>) -> Template {
    let running = *cronframe.running.lock().unwrap();

    let mut active_jobs = vec![];
    let mut timedout_jobs = vec![];
    let mut suspended_jobs = vec![];

    for job in cronframe
        .cron_jobs
        .lock()
        .expect("cron jobs unrwap error in web server")
        .iter()
    {
        let job_type = match job.job {
            CronJobType::Global(_) => CronFilter::Global,
            CronJobType::Function(_) => CronFilter::Function,
            CronJobType::Method(_) => CronFilter::Method,
            CronJobType::CLI => CronFilter::CLI,
        };

        if cronframe.filter.is_none() || cronframe.filter == Some(job_type) {
            if job.status() == "Suspended" {
                suspended_jobs.push(JobList {
                    name: job.name.clone(),
                    id: job.id.to_string(),
                });
            } else if job.status() == "Timed-Out" {
                timedout_jobs.push(JobList {
                    name: job.name.clone(),
                    id: job.id.to_string(),
                });
            } else {
                active_jobs.push(JobList {
                    name: job.name.clone(),
                    id: job.id.to_string(),
                });
            }
        }
    }

    Template::render(
        "index",
        context! {running, active_jobs, timedout_jobs, suspended_jobs},
    )
}

```

```
#[derive(Serialize, Default)]
#[serde(crate = "rocket::serde")]
struct JobInfo {
    name: String,
    id: String,
    r#type: String,
    run_id: String,
    status: String,
    timeout: String,
    schedule: String,
    upcoming_utc: String,
    upcoming_local: String,
    fail: bool,
}

// job page information where it is possilbe to change, schedule, timeout and toggle scheduling suspension
#[get("/job/<name>/<id>")]
fn job_info(name: &str, id: &str, cronframe: &rocket::State<Arc<CronFrame>>) -> Template {
    let running = *cronframe.running.lock().unwrap();
    let mut job_info = JobInfo::default();

    for job in cronframe.cron_jobs.lock().unwrap().iter() {
        if job.name == name && job.id.to_string() == id {
            job_info = JobInfo {
                name: job.name.clone(),
                id: job.id.to_string(),
                r#type: job.type_to_string(),
                run_id: job.run_id(),
                status: job.status(),
                timeout: job.timeout_to_string(),
                schedule: job.schedule(),
                upcoming_utc: job.upcoming_utc(),
                upcoming_local: job.upcoming_local(),
                fail: job.failed,
            };
            break;
        }
    }

    Template::render("job", context! {running, job_info})
}

// API route to change the value of the timeout
#[get("/job/<name>/<id>/toutset/<value>")]
fn update_timeout(name: &str, id: &str, value: i64, cronframe: &rocket::State<Arc<CronFrame>>) {
    for job in cronframe.cron_jobs.lock().unwrap().iter_mut() {
        if job.name == name && job.id.to_string() == id {
            let job_id = format!("{} ID#{}", job.name, job.id);
            job.start_time = None;
            job.set_timeout(value);
```

```
                info!("job @{job_id} - Timeout Update");
            }
        }
    }

    // API route to change the value of the cron expression and therefore the schedule
    #[get("/job/<name>/<id>/schedset/<expression>")]
    fn update_schedule(
        name: &str,
        id: &str,
        expression: &str,
        cronframe: &rocket::State<Arc<CronFrame>>,
    ) {
        for job in cronframe.cron_jobs.lock().unwrap().iter_mut() {
            if job.name == name && job.id.to_string() == id {
                let job_id = format!("{} ID#{}", job.name, job.id);
                if job.set_schedule(expression) {
                    info!("job @{job_id} - Schedule Update");
                } else {
                    info!("job @{job_id} - Schedule Update Fail - Cron Expression Parse Error");
                }
            }
        }
    }

    // API route to toggle the scheduling suspension for a job
    #[get("/job/<name>/<id>/suspension_toggle")]
    fn suspension_handle(name: &str, id: &str, cronframe: &rocket::State<Arc<CronFrame>>) {
        for job in cronframe.cron_jobs.lock().unwrap().iter_mut() {
            if job.name == name && job.id.to_string() == id {
                let job_id = format!("{} ID#{}", job.name, job.id);
                if !job.suspended {
                    job.suspended = true;
                    info!("job @{job_id} - Scheduling Suspended");
                } else {
                    job.suspended = false;
                    info!("job @{job_id} - Scheduling Reprised");
                }
            }
        }
    }

    // API route to start the scheduler
    #[get("/start_scheduler")]
    fn start_scheduler(cronframe: &rocket::State<Arc<CronFrame>>) {
        cronframe.start_scheduler();
    }

    // API route to stop the scheduler
    #[get("/stop_scheduler")]
```

```rust
fn stop_scheduler(cronframe: &rocket::State<Arc<CronFrame>>) {
    cronframe.stop_scheduler();
}

// API route to toggle the scheduling suspension for a job
#[get("/add_cli_job/<expr>/<timeout>/<job>")]
fn add_cli_job(expr: &str, timeout: &str, job: &str, cronframe: &rocket::State<Arc<CronFrame>>) {
    let new_job = JobBuilder::cli_job(job, expr, timeout).build();
    cronframe.add_job(new_job);
}

// API route to stop the scheduler
#[get("/shutdown")]
fn shutdown(cronframe: &rocket::State<Arc<CronFrame>>) {
    cronframe.quit();
}

/// It generates a templates directory either inside the current directory or in the .cronframe directory.
/// The tempaltes directory will contain the following files:
/// - base.html.tera
/// - index.htm.tera
/// - job.html.tera
/// - tingle.js
/// - cronframe.js
/// - styles.css
/// - tingle.css
pub fn generate_template_dir() {
    if std::env::var("CRONFRAME_CLI").is_ok() {
        let home_dir = utils::home_dir();

        if !Path::new(&format!("{home_dir}/.cronframe/templates")).exists() {
            fs::create_dir(format!("{home_dir}/.cronframe/templates"))
                .expect("could not create templates directory");

            let _ = fs::write(
                Path::new(&format!("{home_dir}/.cronframe/templates/base.html.tera")),
                BASE_TEMPLATE,
            );
            let _ = fs::write(
                Path::new(&format!("{home_dir}/.cronframe/templates/index.html.tera")),
                INDEX_TEMPLATE,
            );
            let _ = fs::write(
                Path::new(&format!("{home_dir}/.cronframe/templates/job.html.tera")),
                JOB_TEMPLATE,
            );
            let _ = fs::write(
                Path::new(&format!("{home_dir}/.cronframe/templates/tingle.js")),
                TINGLE_JS,
            );
            let _ = fs::write(
                Path::new(&format!("{home_dir}/.cronframe/templates/cronframe.js")),
                CRONFRAME_JS,
            );
            let _ = fs::write(
                Path::new(&format!("{home_dir}/.cronframe/templates/tingle.css")),
                TINGLE_STYLES,
            );
            let _ = fs::write(
                Path::new(&format!("{home_dir}/.cronframe/templates/styles.css")),
                STYLES,
            );
        }
    } else {
        if !Path::new(&format!("./templates")).exists() {
            fs::create_dir(format!("templates")).expect("could not create templates directory");

            let _ = fs::write(
                Path::new(&format!("./templates/base.html.tera")),
                BASE_TEMPLATE,
            );
            let _ = fs::write(
                Path::new(&format!("./templates/index.html.tera")),
                INDEX_TEMPLATE,
            );
            let _ = fs::write(
                Path::new(&format!("./templates/job.html.tera")),
                JOB_TEMPLATE,
            );
            let _ = fs::write(Path::new(&format!("./templates/tingle.js")), TINGLE_JS);
            let _ = fs::write(
                Path::new(&format!("./templates/cronframe.js")),
                CRONFRAME_JS,
            );
            let _ = fs::write(Path::new(&format!("./templates/tingle.css")), TINGLE_STYLES);
            let _ = fs::write(Path::new(&format!("./templates/styles.css")), STYLES);
        }
    }
    std::thread::sleep(Duration::from_secs(10));
}

// templates folder data: templates/base.tera.html
```

```
const BASE_TEMPLATE: &str = {
    r#"<!DOCTYPE html>
<html class="light-mode">

<head>
    <meta charset="utf-8" />
    <title>CronFrame</title>
    <link href='https://fonts.googleapis.com/css?family=Lato' rel='stylesheet'>
    <link rel="stylesheet" href="/styles">
    <link rel="stylesheet" href="/tingle">

</head>

<body>

    <div id="wrapper">
        <div>
            <div id="barContainer">
                <div id="progressBar">
                    <div id="barStatus"></div>
                </div>
            </div>
            <div id="container">
                <header>
                    <div id="logo">
                        <a href="/"><span class="cron">Cron</span><span style="color:#FF3D00">Frame</span></a>
                    </div>

                    <label class="switch" title="Toggle Color Theme">
                        <input type="checkbox" onchange="toggleMode()" id="slider">
                        <span class="slider"></span>
                    </label>
                    {% if running %}
                    <div id="scheduler_status_running">
                        <span style="padding:0px 5px">ⓘ</span> Scheduler Running
                    </div>
                    <div id="scheduler_stop" onclick="stopScheduler()">
                        <svg height="24" viewBox="0 0 24 20" width="24" xmlns="http://www.w3.org/2000/svg">
                            <path fill="currentColor"
                                d="M12 21c4.411 0 8-3.589 8-8 0-3.35-2.072-6.221-5-7.411v2.223A6 6 0 0 1 18 13c0 3.309-2.691 6-6 6s-6-2.691-6-6a5.999 5.999 0 0 1 3-5.188V5.589C6.072 6.779 4 9.65 4 13c0 4.411 3.589 8 8 8z" />
                            <path fill="currentColor" d="M11 2h2v10h-2z" />
                        </svg>
                    </div>
                    {% else %}
                    <div id="scheduler_status_not_running">
                        <span style="padding:0px 5px">ⓘ</span> Scheduler
Not Running
                    </div>
                    <div id="scheduler_start"
onclick="startScheduler()">
                        <svg height="24" viewBox="0 0 24 20" width="24"
xmlns="http://www.w3.org/2000/svg">
                            <path fill="currentColor"
                                d="M12 21c4.411 0 8-3.589 8-8
0-3.35-2.072-6.221-5-7.411v2.223A6 6 0 0 1 18 13c0 3.309-2.691 6-6
6s-6-2.691-6-6a5.999 5.999 0 0 1 3-5.188V5.589C6.072 6.779 4 9.65 4 13c0
4.411 3.589 8 8 8z" />
                            <path fill="currentColor" d="M11
2h2v10h-2z" />
                        </svg>
                    </div>
                    {% endif %}
                </header>

                <div id="content">
                    {% block content %}
                    {% endblock content %}
                </div>

                <footer>
                    <label class="reload">
                        <input type="checkbox" onchange="toggleReload()"
id="autoreload">
                        <span class="check"></span>
                    </label>
                    5s Reload
                    <a href="https://github.com/antcim/cronframe"
target="_blank" class="repo" title="Developed by Antonio Cimino">
                        <svg xmlns="http://www.w3.org/2000/svg"
width="24" height="24" viewBox="0 0 24 24">
                            <path fill="currentColor"
                                d="M12 0c-6.626 0-12 5.373-12 12 0 5.302
3.438 9.8 8.207
11.387.599.111.793-.261.793-.577v-2.234c-3.338.726-4.033-1.416-4.033-1.416-.546-1.387-1.333-1.7
1.205.084 1.839 1.237 1.839 1.237 1.07 1.834 2.807 1.304
3.492.997.107-.775.418-1.305.762-1.604-2.665-.305-5.467-1.334-5.467-5.931
0-1.311.469-2.381 1.236-3.221-.124-.303-.535-1.524.117-3.176 0 0 1.008-.322
3.301 1.23.957-.266 1.983-.399 3.003-.404 1.02.005 2.047.138 3.006.404
2.291-1.552 3.297-1.23 3.297-1.23.653 1.653.242 2.874.118 3.176.77.84 1.235
1.911 1.235 3.221 0 4.609-2.807 5.624-5.479 5.921.43.372.823 1.102.823
2.222v3.293c0 .319.192.694.801.576 4.765-1.589 8.199-6.086 8.199-11.386
0-6.627-5.373-12-12-12z" />
                        </svg>
                        <span>antcim/cronframe</span>
                    </a>
                </footer>
                <script src="/tinglejs"></script>
                <script src="/cronframejs"></script>
            </div>
        </div>
    </div>
</body>

</html>"#
};

// templates folder data: templates/index.tera.html
const INDEX_TEMPLATE: &str = {
    r#"{% extends "base" %}

{% block content %}
<table id="job_list">
    <tr>
        <th>
            Active Jobs
            <div class="refresh" onclick="reloadPage()">⟳</div>
        </th>
    </tr>
    {% if active_jobs %}
    {% for cron_job in active_jobs %}
    {% set activelink = "/job/" ~ cron_job.name ~ "/" ~ cron_job.id %}
    <tr>
        <td><a href="{{activelink}}">{{cron_job.name}}</a></td>
        <td>{{cron_job.id}}</td>
    </tr>
    {% endfor %}
    {% else %}
    <tr>
        <td>No active job found</td>
    </tr>
    {% endif %}

</table>

<table id="job_list">
    <tr>
        <th>
            Timed-Out Jobs <div class="refresh" onclick="reloadPage()">⟳</div>
        </th>
    </tr>
    {% if timedout_jobs %}
    {% for cron_job in timedout_jobs %}
    {% set timedoutlink = "/job/" ~ cron_job.name ~ "/" ~ cron_job.id %}
    <tr>
        <td><a href="{{timedoutlink}}">{{cron_job.name}}</a></td>
        <td>{{cron_job.id}}</td>
    </tr>
    {% endfor %}
    {% else %}
    <tr>
        <td>No timed-out job found</td>
    </tr>
    {% endif %}
</table>

<table id="job_list">
    <tr>
        <th>
            Suspended Jobs <div class="refresh"
onclick="reloadPage()">⟳</div>
        </th>
    </tr>
    {% if suspended_jobs %}
    {% for cron_job in suspended_jobs %}
    {% set suspendedlink = "/job/" ~ cron_job.name ~ "/" ~ cron_job.id
%}
    <tr>
        <td><a href="{{suspendedlink}}">{{cron_job.name}}</a></td>
        <td>{{cron_job.id}}</td>
    </tr>
    {% endfor %}
    {% else %}
    <tr>
        <td>No suspended job found</td>
    </tr>
    {% endif %}
</table>
{% endblock content %}"#
};

// templates folder data: templates/job.tera.html
const JOB_TEMPLATE: &str = {
    r#"{% extends "base" %}

{% block content %}

{% if job_info.name != ""%}
<table id="job_info">
    <tr>
        <th colspan="2">
            Job Info @{{job_info.name}} <div class="refresh"
onclick="reloadPage()">⟳</div>
        </th>
    </tr>
    <tr>
        <td>Name</td>
        <td colspan="2">{{job_info.name}}</td>
    </tr>
    <tr>
        <td>Id</td>
        <td colspan="2">
            <div class="id_cont">
```

```
                    <span id="job_id">{{job_info.id}}</span>
                    <span class="clipboard"
onclick="copyToClipBoard('job_id')" title="copy to clipboard">
                        <?xml version="1.0" ?>
                        <svg width="16px" height="16px" viewBox="0 0 512
512" xmlns="http://www.w3.org/2000/svg">
                            <path
                                d="M464 0c26.51 0 48 21.49 48 48v288c0
26.51-21.49 48-48 48H176c-26.51 0-48-21.49-48-48V48c0-26.51 21.49-48
48-48h288M176 416c-44.112 0-80-35.888-80-80V128H48c-26.51 0-48 21.49-48
48v288c0 26.51 21.49 48 48 48h288c26.51 0 48-21.49 48-48v-48H176z" />
                        </svg>
                    </span>
                </div>
            </td>
        </tr>
        <tr>
            <td>Type</td>
            <td colspan="2">{{job_info.type}} Job</td>
        </tr>
        {% if job_info.run_id != "None" %}
        <tr>
            <td>Run Id</td>
            <td colspan="2">
                <div class="id_cont">
                    <span id="run_id">{{job_info.run_id}}</span>
                    <span class="clipboard"
onclick="copyToClipBoard('run_id')" title="copy to clipboard">
                        <?xml version="1.0" ?>
                        <svg width="16px" height="16px" viewBox="0 0 512
512" xmlns="http://www.w3.org/2000/svg">
                            <path
                                d="M464 0c26.51 0 48 21.49 48 48v288c0
26.51-21.49 48-48 48H176c-26.51 0-48-21.49-48-48V48c0-26.51 21.49-48
48-48h288M176 416c-44.112 0-80-35.888-80-80V128H48c-26.51 0-48 21.49-48
48v288c0 26.51 21.49 48 48 48h288c26.51 0 48-21.49 48-48v-48H176z" />
                        </svg>
                    </span>
                </div>
            </td>
        </tr>
        {% endif %}
        <tr>
            <td>Status</td>
            <td colspan="">
                {% if job_info.status == "Timed-Out" or job_info.status ==
"Suspended" %}
                <div class="line_status_gray">{{job_info.status}}</div>
                {% elif job_info.status == "Running" %}
                <div class="line_status_green">{{job_info.status}}</div>
                {% else %}
                <div class="line_status_yellow">{{job_info.status}}</div>
                {% endif %}
            </td>
            <td>
                {% if job_info.status != "Suspended" %}
                <button onclick="suspensionHandle()">Suspend Scheduling</button>
                {% else %}
                <button onclick="suspensionHandle()">Reprise Scheduling</button>
                {% endif %}
            </td>
        </tr>
        <tr>
            <td>Fail History</td>
            <td colspan="2">
                {% if job_info.fail %}
                <div class="line_status_orange">Failed instances recorded</div>
                {% else %}
                No failed instances recorded
                {% endif %}
            </td>
        </tr>
        <tr>
            <td>Schedule</td>
            <td>
                {{job_info.schedule}}
            </td>
            <td>
                <input oninput="setSchedule(this.value)" type="text" placeholder="enter cron expression">
                <button onclick="updateSchedule()">Update</button>
            </td>
        </tr>
        <tr>
            <td>Timeout</td>
            <td>
                {% if job_info.timeout != "None" %}
                    {{job_info.timeout | linebreaksbr | split(pat="<br>") | nth(n=1)}}
                    ≈ {{job_info.timeout | linebreaksbr | split(pat="<br>") | nth(n=0)}}
                {% else %}
                    {{job_info.timeout}}
                {% endif %}
            </td>
            <td>
                <input oninput="setTimeout(this.value)" type="number" min="0" placeholder="enter timout in ms">
                <button onclick="updateTimeout()">Update</button>
            </td>
        </tr>
        <tr>
            <td>Upcoming</td>
            <td colspan="2">
                {% if job_info.upcoming_utc == "None due to timeout." %}
                    <p>{{job_info.upcoming_utc}}</p>
                {% else %}
                    <p>{{job_info.upcoming_utc}}</p>
                    {% if job_info.status != "Timed-Out" and 
    job_info.status != "Suspended" %}
                    <p>{{job_info.upcoming_local}} (Local)</p>
                    {% endif %}
                {% endif %}
            </td>
        </tr>
    </table>

    <script>

    </script>

    {% else %}
    <div id="job_info">
        <div class="job_info_item">
            Job not found
        </div>
    </div>
    {% endif %}
    {% endblock content %}"#
    };

    // templates folder data: templates/styles.css
    const STYLES: &str = {
        r#":root {
      --dark-orange: #ff3d00;
      --light-orange: #ffa702;
      --dark-green: #0fa702;
      --green: green;
      --red: red;
    }

    .light-mode {
      --body-bg: #f6f6f6;
      --container-bg: #ffffff;
      --content-bg: #f1f1f1;
      --font-color: #494949;
      --gray-status-color: rgba(0, 0, 0, .3);
      --checkbox: rgba(0, 0, 0, .1);
      --scheduler-status-stop: rgba(0, 0, 0, .1);
      --scheduler-start-stop-text:  rgba(0,0,0,.3);
      --scheduler-status-running: rgba(51, 255, 0, .3);
      --scheduler-status-running-text: rgba(0, 0, 0, .4);
      --scheduler-status-not-running: rgba(237, 233, 157, .7);
      --scheduler-status-not-running-text: rgba(0, 0, 0, .4);
    }

```

```css
.dark-mode {
  --body-bg: #161616;
  --container-bg: #2c2c2c;
  --content-bg: #212121;
  --font-color: #9a9a9a;
  --gray-status-color: rgba(255, 255, 255, .3);
  --checkbox: rgba(0, 0, 0, .3);
  --scheduler-status-stop: rgba(0, 0, 0, .3);
  --scheduler-start-stop-text:  rgba(255,255,255,.3);
  --scheduler-status-running: rgba(51, 255, 0, .2);
  --scheduler-status-running-text: rgba(255, 255, 255, .4);
  --scheduler-status-not-running: rgba(237, 233, 157, .3);
  --scheduler-status-not-running-text: rgba(255, 255, 255, .4);
}

body {
  background: var(--body-bg);
  color: var(--font-color);
  font-family: "Lato"!important;
  margin: 0;
}

a {
  text-decoration: none;
}

a:link {
  color: var(--dark-orange);
}

a:visited {
  color: var(--light-orange);
}

a:hover {
  color: var(--green);
}

a:active {
  color: var(--red);
}

header {
  display: flex;
  padding: 15px;
  align-items: center;
  justify-content: center;
}

footer {
  padding: 15px;
}

```

```
#logo {
  flex: 2;
  font-weight: bold;
  font-size: 30pt;
  margin-right: 50px;
}

.cron {
  color: var(--font-color);
}

.refresh {
  display: inline;
  margin-left: 6px;
}

.refresh:hover {
  cursor: pointer;
  color: var(--light-orange);
}

.refresh:active {
  cursor: pointer;
  color: var(--dark-orange);
}

#wrapper {
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  height: 100vh;
  marign: 0;
  padding: 0;
}

#container {
  display: flex;
  flex-direction: column;
  gap: 5px;
  background: var(--container-bg);
  padding: 10px;
  padding-top: 5px;
  padding-bottomn: 5px;
  border--radius: 6px;
  box-shadow: 0px 0px 3px 0px rgba(0, 0, 0, .1);
  max-width: 1200px;
  min-width: 500px;
}

#scheduler_start, #scheduler_stop{
  color: var(--scheduler-start-stop-text);
  margin-left: 10px;
  font-size: 20pt;
  font-weight: bold;
  cursor: pointer;
  background: var(--scheduler-status-stop);
  border-radius: 6px;
  padding: 5px 8px;
}

#scheduler_start:hover{
  color: var(--scheduler-status-running-text);
  background: var(--scheduler-status-running);
}

#scheduler_start:active{
  color: white;
  background: var(--green);
}

#scheduler_stop:hover{
  color: var(--scheduler-status-running-text);
  background: var(--light-orange);
}

#scheduler_stop:active{
  color: white;
  background: var(--dark-orange);
}

#scheduler_status_running {
  font-weight: bold;
  background: var(--scheduler-status-running);
  padding: 10px;
  border-radius: 6px;
  margin-left: 10px;
  color: var(--scheduler-status-running-text);
}

#scheduler_status_not_running {
  font-weight: bold;
  background: var(--scheduler-status-not-running);
  padding: 10px;
  border-radius: 6px;
  margin-left: 10px;
  color: var(--scheduler-status-not-running-text);
}

#content {
  background: var(--content-bg);
  padding: 10px;
  border-radius: 6px;
  max-height: 70vh;
  overflow: auto;
}
```

```

input[type=text],
input[type=number] {
  padding: 10px;
  display: inline-block;
  border: 1px solid #ccc;
  border-radius: 4px;
  box-sizing: border-box;
}

input[type=text]:focus,
input[type=number]:focus {
  border-color: rgba(229, 103, 23, 0.7);
  box-shadow: 0 1px 1px rgba(229, 103, 23, 0.075) inset, 0 0 4px rgba(229, 103, 23, 0.6);
  outline: 0 none;
}

button {
  background-color: rgba(0, 0, 0, .5);
  font-weight: bold;
  color: white;
  padding: 12px;
  border: none;
  border-radius: 6px;
  cursor: pointer;
  box-shadow: 0 0 2px 1px rgba(0, 0, 0, .1) inset;
}

button:hover {
  background-color: #4caf50;
  color: white;
  border: none;
  cursor: pointer;
  box-shadow: 0 0 2px 1px rgba(0, 0, 0, .1) inset;
}

button:active {
  background-color: var(--dark-orange);
  color: white;
  border: none;
  cursor: pointer;
  box-shadow: 0 0 2px 1px rgba(0, 0, 0, .2) inset;
}

.line_status_green {
  font-weight: bold;
  display: inline-block;
  background: rgba(51, 255, 0, .5);
  padding: 10px;
  border-radius: 6px;
  border: 1px solid rgba(0, 0, 0, .1);
  color: rgba(0, 0, 0, .4);
```

```css
}

.line_status_yellow {
  font-weight: bold;
  display: inline-block;
  background: rgba(255, 236, 102, 1);
  padding: 10px;
  border-radius: 6px;
  border: 1px solid rgba(0, 0, 0, .1);
  color: rgba(0, 0, 0, .4);
}

.line_status_orange {
  font-weight: bold;
  display: inline-block;
  background: rgba(255, 61, 0, .7);
  padding: 10px;
  border-radius: 6px;
  border: 1px solid rgba(0, 0, 0, .1);
  color: rgba(0, 0, 0, .4);
}

.line_status_gray {
  font-weight: bold;
  display: inline-block;
  background: var(--gray-status-color);
  padding: 10px;
  border-radius: 6px;
  border: 1px solid rgba(0, 0, 0, .1);
  color: rgba(0, 0, 0, .4);
}

.clipboard {
  margin: 2px;
  opacity: 0.5;
  filter: invert(50%);
}

.clipboard:hover {
  opacity: 0.8;
  cursor: pointer;
}

.id_cont {
  display: inline;
  padding: 8px;
  border-radius: 6px;
  background: var(--checkbox);
}

table {
  border-collapse: collapse;
}
```

```css

#job_info td {
  padding: 15px;
  border-bottom: 1px solid rgba(0, 0, 0, .05);
}

#job_info th,
#job_info td:nth-child(1) {
  font-weight: bold;
  font-size: 16pt;
  border-right: 1px solid rgba(0, 0, 0, .05);
}

#job_info th {
  text-align: left;
  font-size: 20pt;
  padding: 15px;
  border: 0;
}

#job_info tr:last-child td {
  border: 0px;
  border-right: 1px solid rgba(0, 0, 0, .05);
}

#job_info tr:last-child td:last-child {
  border: 0px;
}

#job_list td {
  padding: 15px;
  border-bottom: 1px solid rgba(0, 0, 0, .05);
}

#job_list th,
#job_list td:nth-child(1) {
  font-weight: bold;
}

#job_list th {
  text-align: left;
  font-size: 20pt;
  padding: 15px;
  border: 0;
}

#job_list tr:last-child td {
  border: 0px;
}

#job_list tr:last-child td:last-child {
  border: 0px;
}
```

```css

.clipboard_toast {
  background: rgba(255, 61, 0, .8);
  color: rgba(0, 0, 0, .4);
  border-radius: 6px;
  top: 0;
  right: 0;
  margin-right: 15px;
  margin-top: 15px;
  position: fixed;
  display: flex;
  flex-direction: row;
  align-items: center;
  padding: 15px;
  gap: 10px;
}

.close_toast {
  font-weight: bold;
  cursor: pointer;
  margin-top: -2px;
}

.close_toast:hover {
  color: white;
}

.switch {
  position: relative;
  display: inline-block;
  width: 45px;
  height: 22px;
}

.switch input {
  opacity: 0;
  width: 0;
  height: 0;
}

.slider {
  position: absolute;
  cursor: pointer;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(237, 233, 157, .1);
  border-radius: 6px;
}

.slider:before {
  position: absolute;
  content: "";
  height: 22px;
  width: 22px;
  margin: auto 0;
  background: silver;
  box-shadow: 0 0 3px 2px rgba(0, 0, 0, .1) inset;
  border-radius: 6px;
}

.switch input:checked + .slider {
  background: rgba(0, 100, 150, .1);
}

.switch input:checked + .slider:before {
  content: "";
  transform: translateX(24px);
  background: var(--light-orange);
  box-shadow: 0 0 3px 2px rgba(255, 0, 0, .1) inset;
}

#barContainer {
  width: 100%;
  padding: 0;
  margin: 0;
}

#progressBar {
  height: 5px;
  background-color: #ddd;
}

#barStatus {
  width: 0%;
  height: 100%;
  background-color: var(--dark-orange);
}

.reload {
  position: relative;
  display: inline-block;
  width: 20px;
  height: 20px;
}

.reload input {
  opacity: 0;
  width: 0;
  height: 0;
}

.check {
  position: absolute;
  cursor: pointer;
```

```css
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: var(--checkbox);
  border-radius: 4px;
}

.check:before {
  position: absolute;
  content: "";
  height: 12px;
  width: 12px;
  margin: auto 0;
  border-radius: 2px;
  top: 4px;
  left: 4px;
}

.reload input:checked + .check:before {
  background: var(--light-orange);
  box-shadow: 0 0 3px 2px rgba(255, 0, 0, .1) inset;
}

::-webkit-scrollbar {
  width: 10px;
}

::-webkit-scrollbar-track {
  background: transparent;
  border-radius: 25px;
}

::-webkit-scrollbar-thumb {
  background: #888;
  border-radius: 25px;
}

::-webkit-scrollbar-thumb:hover {
  background: #555;
}

.repo{
  display: flex;
  gap: 6px;
  float: right;
  color: var(--font-color)!important;
}

.repo:hover{
  color: var(--dark-orange)!important;
}

```

```
.repo:active{
  color: var(--light-orange)!important;
}"#
};

// templates folder data: templates/cronframe.js
const CRONFRAME_JS: &str = {
    r#"// base template scripts

let stopModal = new tingle.modal({
    footer: true,
    stickyFooter: false,
    closeMethods: ['overlay', 'button', 'escape'],
    closeLabel: "Close",
    cssClass: ['custom-class-1', 'custom-class-2'],
    onOpen: function() {
        console.log('modal open');
    },
    onClose: function() {
        console.log('modal closed');
    },
    beforeClose: function() {
        return true; // close the modal
        return false; // nothing happens
    }
});

stopModal.setContent('<h1>Do you want to stop the Scheduler?</h1>');

stopModal.addFooterBtn('No', 'tingle-btn tingle-btn--pull-right tingle-btn', () => {
    stopModal.close();
});

stopModal.addFooterBtn('Yes', 'tingle-btn tingle-btn--pull-right tingle-btn--danger', () => {
    const url = window.location.href + "/stop_scheduler";
    console.log("request to: " + url);
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();
    xhr.responseType = "json";
    xhr.onload = () => {
        if (xhr.readyState == 4 && xhr.status == 200) {
            console.log(xhr.response);
            location.reload();
        } else {
            console.log(`Error: ${xhr.status}`);
        }
    };
});

let startModal = new tingle.modal({
    footer: true,
    stickyFooter: false,
    closeMethods: ['overlay', 'button', 'escape'],
    closeLabel: "Close",
    cssClass: ['custom-class-1', 'custom-class-2'],
    onOpen: function() {
        console.log('modal open');
    },
    onClose: function() {
        console.log('modal closed');
    },
    beforeClose: function() {
        // here's goes some logic
        // e.g. save content before closing the modal
        return true; // close the modal
        return false; // nothing happens
    }
});

startModal.setContent('<h1>Do you want to start the Scheduler?</h1>');

startModal.addFooterBtn('No', 'tingle-btn tingle-btn--pull-right
tingle-btn', () => {
    startModal.close();
});

startModal.addFooterBtn('Yes', 'tingle-btn tingle-btn--pull-right
tingle-btn--danger', () => {
    const url = window.location.href + "/start_scheduler";
    console.log("request to: " + url);
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();
    xhr.responseType = "json";
    xhr.onload = () => {
        if (xhr.readyState == 4 && xhr.status == 200) {
            console.log(xhr.response);
            location.reload();
        } else {
            console.log(`Error: ${xhr.status}`);
        }
    };
});

let barWidth = 0;

document.getElementById("barContainer").style.width =
document.getElementById("container").style.width;

const advanceBar = () => {
    if (barWidth < 100) {
        barWidth = barWidth + 3.125;
        document.getElementById("barStatus").style.width = barWidth +
'%';
    }
};

const reloadPage = () => {
    location.reload();
};

const setTheme = (value) => {
    localStorage.setItem('mode', value);
    document.documentElement.className = value;
};

const setAutoreload = (value) => {
    localStorage.setItem('autoreload', value);
    document.documentElement.className = value;
    reloadPage();
};

const toggleMode = () => {
    if (localStorage.getItem('mode') === 'dark-mode') {
        setTheme('light-mode');
    } else {
        setTheme('dark-mode');
    }
};

const toggleReload = () => {
    if (localStorage.getItem('autoreload') === 'yes') {
        setAutoreload('no');
    } else {
        setAutoreload('yes');
    }
};

const init = () => {
    setupTheme();
    setupBar();
};

const setupTheme = () => {
    if (localStorage.getItem('mode') === 'dark-mode') {
        setTheme('dark-mode');
        document.getElementById('slider').checked = false;
    } else {
        setTheme('light-mode');
        document.getElementById('slider').checked = true;
    }
};

const setupBar = () => {
    if (localStorage.getItem('autoreload') === 'yes') {
        setInterval(reloadPage, 5000);
        setInterval(advanceBar, 125);
        document.getElementById('autoreload').checked = true;
    } else {
        document.getElementById("barStatus").style.width = '100%';
        document.getElementById('autoreload').checked = false;
    }
};

init();

const startScheduler = () => {
    startModal.open();
}

const stopScheduler = () => {
    stopModal.open();
}

// job page scripts

let timeout = 0;
    let schedule = "* * * * * * *";

    const setTimeout = (value) => {
        console.log(value);
        timeout = value
    };

    const updateTimeout = () => {
        console.log("request to: " + window.location.href + "/toutset/"
+ timeout);
        const xhr = new XMLHttpRequest();
        xhr.open("GET", window.location.href + "/toutset/" + timeout);
        xhr.send();
        xhr.responseType = "json";
        xhr.onload = () => {
            if (xhr.readyState == 4 && xhr.status == 200) {
                console.log(xhr.response);
                location.reload();
            } else {
                console.log(`Error: ${xhr.status}`);
            }
        };
    }

    const setSchedule = (value) => {
        console.log(value);
        schedule = value
    };

    const updateSchedule = () => {
        console.log("request to: " + window.location.href + "/
schedset/" + schedule);
```

```
        const xhr = new XMLHttpRequest();
        xhr.open("GET", window.location.href + "/schedset/" +
schedule.replace("/", "slh"));
        xhr.send();
        xhr.responseType = "json";
        xhr.onload = () => {
            if (xhr.readyState == 4 && xhr.status == 200) {
                console.log(xhr.response);
                location.reload();
            } else {
                console.log(`Error: ${xhr.status}`);
            }
        };
    }

    const suspensionHandle = () => {
        console.log("request to: " + window.location.href + "/
suspension_toggle");
        const xhr = new XMLHttpRequest();
        xhr.open("GET", window.location.href + "/suspension_toggle");
        xhr.send();
        xhr.responseType = "json";
        xhr.onload = () => {
            if (xhr.readyState == 4 && xhr.status == 200) {
                console.log(xhr.response);
                location.reload();
            } else {
                console.log(`Error: ${xhr.status}`);
            }
        };
    }

    const copyToClipBoard = (element) => {
        var copyText = document.getElementById(element);
        navigator.clipboard.writeText(copyText.innerHTML);
        toast("Copied to Clipboard");
    }

    let notify_shown = false;

    const toast = (text) => {
        if (notify_shown) return;

        notify_shown = true;

        var toast = document.createElement('div');
        toast.className = "clipboard_toast";

        var message = document.createElement("div");
        message.textContent = text;
        toast.appendChild(message);

        var close = document.createElement("div");
        close.className = "close_toast";
        close.innerHTML = "x"
        close.addEventListener("click", () => {
            toast.remove();
            notify_shown = false;
        })
        toast.append(close);

        document.body.appendChild(toast);

        window.setTimeout(() => {
            toast.remove();
            notify_shown = false;
        }, 3000);
    }"#
};

// templates folder data: templates/tingle.js
const TINGLE_JS: &str = {
    r#"/**
 * tingle.js - A simple modal plugin written in pure JavaScript
 * @version v0.16.0
 * @link https://github.com/robinparisi/tingle#readme
 * @license MIT
 */

/* global define, module */
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    define(factory)
  } else if (typeof exports === 'object') {
    module.exports = factory()
  } else {
    root.tingle = factory()
  }
}(this, function () {
  /* ----------------------------------------------------------- */
  /* == modal */
  /* ----------------------------------------------------------- */

  var isBusy = false

  function Modal (options) {
    var defaults = {
      onClose: null,
      onOpen: null,
      beforeOpen: null,
      beforeClose: null,
      stickyFooter: false,
      footer: false,
      cssClass: [],
      closeLabel: 'Close',
      closeMethods: ['overlay', 'button', 'escape']
    }

    // extends config
    this.opts = extend({}, defaults, options)

    // init modal
    this.init()
  }

  Modal.prototype.init = function () {
    if (this.modal) {
      return
    }

    _build.call(this)
    _bindEvents.call(this)

    // insert modal in dom
    document.body.appendChild(this.modal, document.body.firstChild)

    if (this.opts.footer) {
      this.addFooter()
    }

    return this
  }

  Modal.prototype._busy = function (state) {
    isBusy = state
  }

  Modal.prototype._isBusy = function () {
    return isBusy
  }

  Modal.prototype.destroy = function () {
    if (this.modal === null) {
      return
    }

    // restore scrolling
    if (this.isOpen()) {
      this.close(true)
    }

    // unbind all events
    _unbindEvents.call(this)

    // remove modal from dom
    this.modal.parentNode.removeChild(this.modal)

    this.modal = null
  }
```

```

Modal.prototype.isOpen = function () {
  return !!this.modal.classList.contains('tingle-modal--visible')
}

Modal.prototype.open = function () {
  if (this._isBusy()) return
  this._busy(true)

  var self = this

  // before open callback
  if (typeof self.opts.beforeOpen === 'function') {
    self.opts.beforeOpen()
  }

  if (this.modal.style.removeProperty) {
    this.modal.style.removeProperty('display')
  } else {
    this.modal.style.removeAttribute('display')
  }

  // prevent text selection when opening multiple times
  document.getSelection().removeAllRanges()

  // prevent double scroll
  this._scrollPosition = window.pageYOffset
  document.body.classList.add('tingle-enabled')
  document.body.style.top = -this._scrollPosition + 'px'

  // sticky footer
  this.setStickyFooter(this.opts.stickyFooter)

  // show modal
  this.modal.classList.add('tingle-modal--visible')

  // onOpen callback
  if (typeof self.opts.onOpen === 'function') {
    self.opts.onOpen.call(self)
  }

  self._busy(false)

  // check if modal is bigger than screen height
  this.checkOverflow()

  return this
}

Modal.prototype.close = function (force) {
  if (this._isBusy()) return
  this._busy(true)
  force = force || false
```

```

    //  before close
    if (typeof this.opts.beforeClose === 'function') {
      var close = this.opts.beforeClose.call(this)
      if (!close) {
        this._busy(false)
        return
      }
    }

    document.body.classList.remove('tingle-enabled')
    document.body.style.top = null
    window.scrollTo({
      top: this._scrollPosition,
      behavior: 'instant'
    })

    this.modal.classList.remove('tingle-modal--visible')

    // using similar setup as onOpen
    var self = this

    self.modal.style.display = 'none'

    // onClose callback
    if (typeof self.opts.onClose === 'function') {
      self.opts.onClose.call(this)
    }

    // release modal
    self._busy(false)
  }

  Modal.prototype.setContent = function (content) {
    // check type of content : String or Node
    if (typeof content === 'string') {
      this.modalBoxContent.innerHTML = content
    } else {
      this.modalBoxContent.innerHTML = ''
      this.modalBoxContent.appendChild(content)
    }

    if (this.isOpen()) {
      // check if modal is bigger than screen height
      this.checkOverflow()
    }

    return this
  }

  Modal.prototype.getContent = function () {
    return this.modalBoxContent
  }

  Modal.prototype.addFooter = function () {
    // add footer to modal
    _buildFooter.call(this)

    return this
  }

  Modal.prototype.setFooterContent = function (content) {
    // set footer content
    this.modalBoxFooter.innerHTML = content

    return this
  }

  Modal.prototype.getFooterContent = function () {
    return this.modalBoxFooter
  }

  Modal.prototype.setStickyFooter = function (isSticky) {
    // if the modal is smaller than the viewport height, we don't need sticky
    if (!this.isOverflow()) {
      isSticky = false
    }

    if (isSticky) {
      if (this.modalBox.contains(this.modalBoxFooter)) {
        this.modalBox.removeChild(this.modalBoxFooter)
        this.modal.appendChild(this.modalBoxFooter)
        this.modalBoxFooter.classList.add('tingle-modal-box__footer--sticky')
        _recalculateFooterPosition.call(this)
      }
      this.modalBoxContent.style['padding-bottom'] = this.modalBoxFooter.clientHeight + 20 + 'px'
    } else if (this.modalBoxFooter) {
      if (!this.modalBox.contains(this.modalBoxFooter)) {
        this.modal.removeChild(this.modalBoxFooter)
        this.modalBox.appendChild(this.modalBoxFooter)
        this.modalBoxFooter.style.width = 'auto'
        this.modalBoxFooter.style.left = ''
        this.modalBoxContent.style['padding-bottom'] = ''
        this.modalBoxFooter.classList.remove('tingle-modal-box__footer--sticky')
      }
    }

    return this
  }

  Modal.prototype.addFooterBtn = function (label, cssClass, callback) {
    var btn = document.createElement('button')

    // set label
    btn.innerHTML = label

    // bind callback
    btn.addEventListener('click', callback)

    if (typeof cssClass === 'string' && cssClass.length) {
      // add classes to btn
      cssClass.split(' ').forEach(function (item) {
        btn.classList.add(item)
      })
    }

    this.modalBoxFooter.appendChild(btn)

    return btn
  }

  Modal.prototype.resize = function () {
    // eslint-disable-next-line no-console
    console.warn('Resize is deprecated and will be removed in version
  1.0')
  }

  Modal.prototype.isOverflow = function () {
    var viewportHeight = window.innerHeight
    var modalHeight = this.modalBox.clientHeight

    return modalHeight >= viewportHeight
  }

  Modal.prototype.checkOverflow = function () {
    // only if the modal is currently shown
    if (this.modal.classList.contains('tingle-modal--visible')) {
      if (this.isOverflow()) {
        this.modal.classList.add('tingle-modal--overflow')
      } else {
        this.modal.classList.remove('tingle-modal--overflow')
      }

      if (!this.isOverflow() && this.opts.stickyFooter) {
        this.setStickyFooter(false)
      } else if (this.isOverflow() && this.opts.stickyFooter) {
        _recalculateFooterPosition.call(this)
        this.setStickyFooter(true)
      }
    }
  }

  /* ----------------------------------------------------------- */
  /* == private methods */
  /* ----------------------------------------------------------- */
```

```

  function closeIcon () {
    return '<svg viewBox="0 0 10 10" xmlns="http://www.w3.org/2000/
svg"><path d="M.3 9.7c.2.2.4.3.7.3.3 0 .5-.1.7-.3L5 6.4l3.3
3.3c.2.2.5.3.7.3.2 0 .5-.1.7-.3.4-.4.4-1 0-1.4L6.4 5l3.3-3.3c.4-.4.4-1
0-1.4-.4-.4-1-.4-1.4 0L5 3.6 1.7.3C1.3-.1.7-.1.3.3c-.4.4-.4 1 0 1.4L3.6 5 .3
8.3c-.4.4-.4 1 0 1.4z" fill="black" fill-rule="nonzero"/></svg>'
  }

  function _recalculateFooterPosition () {
    if (!this.modalBoxFooter) {
      return
    }
    this.modalBoxFooter.style.width = this.modalBox.clientWidth + 'px'
    this.modalBoxFooter.style.left = this.modalBox.offsetLeft + 'px'
  }

  function _build () {
    // wrapper
    this.modal = document.createElement('div')
    this.modal.classList.add('tingle-modal')

    // remove cusor if no overlay close method
    if (this.opts.closeMethods.length === 0 ||
this.opts.closeMethods.indexOf('overlay') === -1) {
      this.modal.classList.add('tingle-modal--noOverlayClose')
    }

    this.modal.style.display = 'none'

    // custom class
    this.opts.cssClass.forEach(function (item) {
      if (typeof item === 'string') {
        this.modal.classList.add(item)
      }
    }, this)

    // close btn
    if (this.opts.closeMethods.indexOf('button') !== -1) {
      this.modalCloseBtn = document.createElement('button')
      this.modalCloseBtn.type = 'button'
      this.modalCloseBtn.classList.add('tingle-modal__close')

      this.modalCloseBtnIcon = document.createElement('span')
      this.modalCloseBtnIcon.classList.add('tingle-modal__closeIcon')
      this.modalCloseBtnIcon.innerHTML = closeIcon()

      this.modalCloseBtnLabel = document.createElement('span')
      this.modalCloseBtnLabel.classList.add('tingle-modal__closeLabel')
      this.modalCloseBtnLabel.innerHTML = this.opts.closeLabel

      this.modalCloseBtn.appendChild(this.modalCloseBtnIcon)
      this.modalCloseBtn.appendChild(this.modalCloseBtnLabel)
    }

    // modal
    this.modalBox = document.createElement('div')
    this.modalBox.classList.add('tingle-modal-box')

    // modal box content
    this.modalBoxContent = document.createElement('div')
    this.modalBoxContent.classList.add('tingle-modal-box__content')

    this.modalBox.appendChild(this.modalBoxContent)

    if (this.opts.closeMethods.indexOf('button') !== -1) {
      this.modal.appendChild(this.modalCloseBtn)
    }

    this.modal.appendChild(this.modalBox)
  }

  function _buildFooter () {
    this.modalBoxFooter = document.createElement('div')
    this.modalBoxFooter.classList.add('tingle-modal-box__footer')
    this.modalBox.appendChild(this.modalBoxFooter)
  }

  function _bindEvents () {
    this._events = {
      clickCloseBtn: this.close.bind(this),
      clickOverlay: _handleClickOutside.bind(this),
      resize: this.checkOverflow.bind(this),
      keyboardNav: _handleKeyboardNav.bind(this)
    }

    if (this.opts.closeMethods.indexOf('button') !== -1) {
      this.modalCloseBtn.addEventListener('click',
this._events.clickCloseBtn)
    }

    this.modal.addEventListener('mousedown', this._events.clickOverlay)
    window.addEventListener('resize', this._events.resize)
    document.addEventListener('keydown', this._events.keyboardNav)
  }

  function _handleKeyboardNav (event) {
    // escape key
    if (this.opts.closeMethods.indexOf('escape') !== -1 && event.which
=== 27 && this.isOpen()) {
      this.close()
    }
  }

  function _handleClickOutside (event) {
    // on macOS, click on scrollbar (hidden mode) will trigger close
```

```
event so we need to bypass this behavior by detecting scrollbar mode
    var scrollbarWidth = this.modal.offsetWidth -
this.modal.clientWidth
    var clickedOnScrollbar = event.clientX >= this.modal.offsetWidth -
15 // 15px is macOS scrollbar default width
    var isScrollable = this.modal.scrollHeight !==
this.modal.offsetHeight
    if (navigator.platform === 'MacIntel' && scrollbarWidth === 0 &&
clickedOnScrollbar && isScrollable) {
      return
    }

    // if click is outside the modal
    if (this.opts.closeMethods.indexOf('overlay') !== -1 && !
_findAncestor(event.target, 'tingle-modal') &&
        event.clientX < this.modal.clientWidth) {
      this.close()
    }
  }

  function _findAncestor (el, cls) {
    while ((el = el.parentElement) && !el.classList.contains(cls));
    return el
  }

  function _unbindEvents () {
    if (this.opts.closeMethods.indexOf('button') !== -1) {
      this.modalCloseBtn.removeEventListener('click',
this._events.clickCloseBtn)
    }
    this.modal.removeEventListener('mousedown',
this._events.clickOverlay)
    window.removeEventListener('resize', this._events.resize)
    document.removeEventListener('keydown', this._events.keyboardNav)
  }

  /* ----------------------------------------------------------- */
  /* == helpers */
  /* ----------------------------------------------------------- */

  function extend () {
    for (var i = 1; i < arguments.length; i++) {
      for (var key in arguments[i]) {
        if (arguments[i].hasOwnProperty(key)) {
          arguments[0][key] = arguments[i][key]
        }
      }
    }
    return arguments[0]
  }

  /* ----------------------------------------------------------- */
  /* == return */
  /* ----------------------------------------------------------- */

  return {
    modal: Modal
  }
}))
"#
};

// templates folder data: templates/tingle.css
const TINGLE_STYLES: &str = {
    r#"/**
 * tingle.js - A simple modal plugin written in pure JavaScript
 * @version v0.16.0
 * @link https://github.com/robinparisi/tingle#readme
 * @license MIT
 */

// modified for cronframe

.tingle-modal * {
  box-sizing: border-box;
}

.tingle-modal {
  position: fixed;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  z-index: 1000;
  display: flex;
  visibility: hidden;
  flex-direction: column;
  align-items: center;
  overflow: hidden;
  -webkit-overflow-scrolling: touch;
  background: rgba(0, 0, 0, .9);
  opacity: 0;
  cursor: url("data:image/svg+xml,%3Csvg width='19' height='19' xmlns='http://www.w3.org/2000/svg'%3E%3Cpath d='M15.514.535l-6.42 6.42L2.677.536a1.517 1.517 0 00-2.14 0 1.517 1.517 0 000 2.14l6.42 6.419-6.42 6.419a1.517 1.517 0 000 2.14 1.517 1.517 0 002.14 0l6.419-6.42 6.419 6.42a1.517 1.517 0 002.14 0 1.517 1.517 0 000-2.14l-6.42-6.42 6.42-6.418a1.517 1.517 0 000-2.14 1.516 1.516 0 00-2.14 0z' fill='%23FFF' fill-rule='nonzero'/%3E%3C/svg%3E"), auto;
}

@supports ((-webkit-backdrop-filter: blur(12px)) or (backdrop-filter: blur(12px))) {
  .tingle-modal {
    -webkit-backdrop-filter: blur(12px);
    backdrop-filter: blur(12px);
  }
}

/* confirm and alerts
-------------------------------------------------------------- */

.tingle-modal--confirm .tingle-modal-box {
  text-align: center;
}

/* modal
-------------------------------------------------------------- */

.tingle-modal--noOverlayClose {
  cursor: default;
}

.tingle-modal--noClose .tingle-modal__close {
  display: none;
}

.tingle-modal__close {
  position: fixed;
  top: 2.5rem;
  right: 2.5rem;
  z-index: 1000;
  padding: 0;
  width: 2rem;
  height: 2rem;
  border: none;
  background-color: transparent;
  color: #fff;
  cursor: pointer;
}

.tingle-modal__close svg * {
  fill: currentColor;
}

.tingle-modal__closeLabel {
  display: none;
}

.tingle-modal__close:hover {
  color: var(--light-orange);
  background: transparent;
}

.tingle-modal__close:active {
  color: var(--dark-orange);
  background: transparent;
}

```

```css
.tingle-modal-box {
  max-width: 600px;
  position: relative;
  flex-shrink: 0;
  margin-top: auto;
  margin-bottom: auto;
  width: 60%;
  border-radius: 4px;
  background: var(--container-bg);
  opacity: 1;
  cursor: auto;
  will-change: transform, opacity;
}

.tingle-modal-box__content {
  padding: 3rem 3rem;
}

.tingle-modal-box__footer {
  padding: 1.5rem 2rem;
  width: auto;
  border-bottom-right-radius: 4px;
  border-bottom-left-radius: 4px;
  background-color: var(--content-bg);
  cursor: auto;
}

.tingle-modal-box__footer::after {
  display: table;
  clear: both;
  content: "";
}

.tingle-modal-box__footer--sticky {
  position: fixed;
  bottom: -200px; /* TODO : find a better way */
  z-index: 10001;
  opacity: 1;
  transition: bottom .3s ease-in-out .3s;
}

/* state
-------------------------------------------------------------- */

.tingle-enabled {
  position: fixed;
  right: 0;
  left: 0;
  overflow: hidden;
}

.tingle-modal--visible .tingle-modal-box__footer {
  bottom: 0;
```

```css
}

.tingle-modal--visible {
  visibility: visible;
  opacity: 1;
}

.tingle-modal--visible .tingle-modal-box {
  animation: scale .2s cubic-bezier(.68, -.55, .265, 1.55) forwards;
}

.tingle-modal--overflow {
  overflow-y: scroll;
  padding-top: 8vh;
}

/* btn
-------------------------------------------------------------- */

.tingle-btn {
  display: inline-block;
  margin: 0 .5rem;
  padding: 1rem 2rem;
  border: none;
  background-color: grey;
  box-shadow: none;
  color: #fff;
  vertical-align: middle;
  text-decoration: none;
  font-size: inherit;
  font-family: inherit;
  line-height: normal;
  cursor: pointer;
  transition: background-color .4s ease;
}

.tingle-btn--primary {
  background-color: #3498db;
}

.tingle-btn--danger {
  background-color: var(--light-orange);
}

.tingle-btn--default {
  background-color: #34495e;
}

.tingle-btn--pull-left {
  float: left;
}

.tingle-btn--pull-right {
  float: right;
}

/* responsive
-------------------------------------------------------------- */

@media (max-width : 540px) {
  .tingle-modal {
    top: 0px;
    display: block;
    padding-top: 60px;
    width: 100%;
  }

  .tingle-modal-box {
    width: auto;
    border-radius: 0;
  }

  .tingle-modal-box__content {
    overflow-y: scroll;
  }

  .tingle-modal--noClose {
    top: 0;
  }

  .tingle-modal--noOverlayClose {
    padding-top: 0;
  }

  .tingle-modal-box__footer .tingle-btn {
    display: block;
    float: none;
    margin-bottom: 1rem;
    width: 100%;
  }

  .tingle-modal__close {
    top: 0;
    right: 0;
    left: 0;
    display: block;
    width: 100%;
    height: 60px;
    border: none;
    background-color: var(--content-bg);
    box-shadow: none;
    color: #fff;
  }

  .tingle-modal__closeLabel {
    display: inline-block;
    vertical-align: middle;
    font-size: 1.6rem;
    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI",
"Roboto", "Oxygen", "Ubuntu", "Cantarell", "Fira Sans", "Droid Sans",
"Helvetica Neue", sans-serif;
  }

  .tingle-modal__closeIcon {
    display: inline-block;
    margin-right: .8rem;
    width: 1.6rem;
    vertical-align: middle;
    font-size: 0;
  }
}

/* animations
-------------------------------------------------------------- */

@keyframes scale {
  0% {
    opacity: 0;
    transform: scale(.9);
  }
  100% {
    opacity: 1;
    transform: scale(1);
  }
}
"#
};
```

A-7 - src/logger.rs

```
//! Default logger setup for the cronframe framework and the testing suite

use crate::{config::read_config, utils};
use chrono::Duration;
use log4rs::{
    append::{
        file::FileAppender,
        rolling_file::{
            policy::compound::{
                roll::fixed_window::FixedWindowRoller,
trigger::size::SizeTrigger, CompoundPolicy,
            },
            RollingFileAppender,
        },
    },
    config::{Appender, Config, Root},
    encode::pattern::PatternEncoder,
};
```

```rust

/// this logger configuration is used for testing
pub fn appender_logger(log_file: &str) -> log4rs::Handle {
    let pattern = "{d(%Y-%m-%d %H:%M:%S %Z)} {l} {t} - {m}{n}";

    let log_file = FileAppender::builder()
        .encoder(Box::new(PatternEncoder::new(pattern)))
        .append(false)
        .build(log_file)
        .expect("appender_logger log file unwrap error");

    let config = Config::builder()
        .appender(Appender::builder().build("log_file", Box::new(log_file)))
        .build(
            Root::builder()
                .appender("log_file")
                .build(log::LevelFilter::Info),
        )
        .expect("appender_logger config unwrap error");

    log4rs::init_config(config).expect("appender_logger init error")
}

/// this is used to change the log file for each new test
pub fn appender_config(log_file: &str) -> log4rs::Config {
    let pattern = "{d(%Y-%m-%d %H:%M:%S %Z)} {l} {t} - {m}{n}";

    let log_file = FileAppender::builder()
        .encoder(Box::new(PatternEncoder::new(pattern)))
        .append(false)
        .build(log_file)
        .expect("appender_config log file unwrap error");

    Config::builder()
        .appender(Appender::builder().build("log_file", Box::new(log_file)))
        .build(
            Root::builder()
                .appender("log_file")
                .build(log::LevelFilter::Info),
        )
        .expect("appender_logger config unwrap error")
}

/// this sets the logger from either the default configuration or from the toml file
pub fn rolling_logger() -> log4rs::Handle {
    let mut window_size = 3;
    let mut size_limit = 1000 * 1024;
    let mut log_dir = "log".to_string();
    let mut latest_file_name = "latest".to_string();
    let mut archive_file_name = "archive".to_string();
    let mut pattern = "{d(%Y-%m-%d %H:%M:%S %Z)} {l} {t} - {m}{n}".to_string();
    let mut level_filter = log::LevelFilter::Info;

    if let Some(config_data) = read_config() {
        if let Some(logger_data) = config_data.logger {
            if let Some(data) = logger_data.archive_files {
                window_size = data;
            }
            if let Some(data) = logger_data.file_size {
                size_limit = size_limit * data;
            }
            if let Some(data) = logger_data.dir {
                log_dir = data;
            }
            if let Some(data) = logger_data.latest_file_name {
                latest_file_name = data;
            }
            if let Some(data) = logger_data.archive_file_name {
                archive_file_name = data;
            }
            if let Some(data) = logger_data.msg_pattern {
                pattern = data;
            }
            if let Some(data) = logger_data.level_filter {
                match data.as_str() {
                    "off" => level_filter = log::LevelFilter::Off,
                    "error" => level_filter = log::LevelFilter::Error,
                    "warn" => level_filter = log::LevelFilter::Warn,
                    "debug" => level_filter = log::LevelFilter::Debug,
                    _ => (),
                }
            }
        }
    };

    if std::env::var("CRONFRAME_CLI").is_ok() {
        let home_dir = utils::home_dir();
        log_dir = format!("{home_dir}/.cronframe/log");
    }

    let archive_file = format!("{log_dir}/{archive_file_name}.log").replace(".log", "_{}.log");

    // retain latest and archive logfiles at restart as per rolling policy
    if !std::path::Path::new(&format!("{log_dir}/{latest_file_name}")).exists() {
        let _ = std::fs::remove_file(format!(
            "./{log_dir}/{archive_file_name}_{}.log",
            window_size - 1
        ));

```

```rust
        for i in (1..=(window_size - 1)).rev() {
            let _ = std::fs::rename(
                format!("{log_dir}/{archive_file_name}_{}.log", i - 1),
                format!("{log_dir}/{archive_file_name}_{}.log", i),
            );
        }

        let _ = std::fs::rename(
            format!("{log_dir}/{latest_file_name}.log"),
            format!("{log_dir}/{archive_file_name}_0.log"),
        );

        std::thread::sleep(Duration::seconds(5).to_std().unwrap());
    }

    let roller = FixedWindowRoller::builder()
        .build(&archive_file, window_size)
        .unwrap();

    let trigger = SizeTrigger::new(size_limit);

    let policy = CompoundPolicy::new(Box::new(trigger),
Box::new(roller));

    let log_file = RollingFileAppender::builder()
        .encoder(Box::new(PatternEncoder::new(&pattern)))
        .append(false)
        .build(
            &format!("{log_dir}/{latest_file_name}.log"),
            Box::new(policy),
        )
        .expect("rolling_logger log file unwrap error");

    let config = Config::builder()
        .appender(Appender::builder().build("log_file",
Box::new(log_file)))
        .build(Root::builder().appender("log_file").build(level_filter))
        .expect("rolling_logger config unwrap error");

    log4rs::init_config(config).expect("rolling_logger init error")
}

```

A-8 - src/config.rs

```rust
//! Configuration avaliable in `cronframe.toml`

use crate::utils;
use rocket::serde::Deserialize;
use std::fs;
use toml;

```

```rust
#[derive(Deserialize)]
#[serde(crate = "rocket::serde")]
pub struct ConfigData {
    pub webserver: Option<ServerConfig>,
    pub logger: Option<LoggerConfig>,
    pub scheduler: Option<SchedulerConfig>,
}

#[derive(Deserialize)]
#[serde(crate = "rocket::serde")]
pub struct ServerConfig {
    pub port: Option<u16>,
    pub ip: Option<String>,
}

#[derive(Deserialize)]
#[serde(crate = "rocket::serde")]
pub struct LoggerConfig {
    pub dir: Option<String>,
    pub file_size: Option<u64>,
    pub archive_files: Option<u32>,
    pub latest_file_name: Option<String>,
    pub archive_file_name: Option<String>,
    pub msg_pattern: Option<String>,
    pub level_filter: Option<String>,
}

#[derive(Deserialize)]
#[serde(crate = "rocket::serde")]
pub struct SchedulerConfig {
    pub grace: Option<u32>,
}

/// This function reads cronframe configuration data from the `cronframe.toml` file.
///
/// There are three sections to the configuration:
/// - webserver
/// - logger
/// - scheduler
///
///
/// [webserver]
/// port = 8098
///
/// [logger]
/// dir = "log"
/// file_size = 1 # this is in MB
/// archive_files = 3
/// latest_file_name = "latest"
/// archive_file_name = "archive"
/// msg_pattern = "{l} {t} - {m}{n}"
/// level_filter = "info"
```

```rust
///
/// [scheduler]
/// grace = 250 # this is in ms
///
///
pub fn read_config() -> Option<ConfigData> {
    let filename = if std::env::var("CRONFRAME_CLI").is_ok() {
        let home_dir = utils::home_dir();
        &format!("{home_dir}/.cronframe/cronframe.toml")
    } else {
        "cronframe.toml"
    };

    if let Ok(file_content) = fs::read_to_string(filename) {
        if let Ok(data) = toml::from_str(&file_content) {
            data
        } else {
            error!("cronframe.toml - data read error");
            None
        }
    } else {
        info!("cronframe.toml - file not found");
        None
    }
}
```

A-9 - src/utils.rs

```rust
//! Utilities

use crate::config::read_config;
use chrono::{DateTime, Local, Utc};

/// Convertion from UTC to Local time
pub fn local_time(utc_time: DateTime<Utc>) -> DateTime<Local> {
    let local_time: DateTime<Local> = DateTime::from(utc_time);
    local_time
}

pub fn home_dir() -> String {
    let tmp = home::home_dir().unwrap();
    tmp.to_str().unwrap().to_owned()
}

pub fn ip_and_port() -> (String, u16) {
    match read_config() {
        Some(config_data) => {
            if let Some(webserver_data) = config_data.webserver {
                (
                    webserver_data.ip.unwrap_or_else(||
"127.0.0.1".to_string()),
                    webserver_data.port.unwrap_or_else(|| 8098),
                )
            } else {
                ("localhost".to_string(), 8098)
            }
        }
        None => ("localhost".to_string(), 8098),
    }
}
```

A-10 - src/bin.rs

```
//! CronFrame CLI Tool v0.1.3
//! Use the cronframe help comamnd for details

use clap::{arg, command};
use colored::*;
use cronframe::{
    utils::{self, ip_and_port},
    web_server, CronFilter, CronFrame,
};
use std::{
    fs,
    io::BufRead,
    path::Path,
    process::{Command, Stdio},
};

fn main() {
    std::env::set_var("CRONFRAME_CLI", "true");

    // cli args parsing
    let matches = command!()
        .propagate_version(true)
        .subcommand_required(true)
        .arg_required_else_help(true)
        // cronframe start
        .subcommand(
            clap::Command::new("start")
                .about("Start the CronFrame Webserver and Job Scheduler
in background."),
        )
        // cronframe run
        .subcommand(
            clap::Command::new("run")
                .about("Run the CronFrame Webserver and Job Scheduler in
the terminal."),
        )
        // cronframe add EXPR TIMEOUT JOB
        .subcommand(
            clap::Command::new("add")
                .about("Adds a new cli job to a CronFrame instance.")
                .args(&[
```

```rust
                    arg!([EXPR] "The Cron Expression to use for job scheduling."),
                    arg!([TIMEOUT] "The value in ms to use for the timeout."),
                    arg!([JOB] "The path containing the source code of the job."),
                ])
                .arg_required_else_help(true)
                .arg(
                    arg!(-p --port <VALUE>)
                        .required(false)
                        .action(clap::ArgAction::Set),
                ),
        )
        // cronframe load
        .subcommand(
            clap::Command::new("load")
                .about("Load jobs from definition file.")
                .arg(
                    arg!(-f --file <PATH>)
                        .required(false)
                        .action(clap::ArgAction::Set),
                ),
        )
        // cronframe scheduler ACTION
        .subcommand(
            clap::Command::new("scheduler")
                .about("Perform actions on the scheduler like start and stop")
                .args(&[arg!([ACTION] "Action to perform = (start, stop)")])
                .arg_required_else_help(true)
                .arg(
                    arg!(-p --port <VALUE>)
                        .required(false)
                        .action(clap::ArgAction::Set),
                ),
        )
        // cronframe shutdown
        .subcommand(
            clap::Command::new("shutdown")
                .about("Shutdown the CronFrame Webserver and Job Scheduler."),
        )
        .get_matches();

    match matches.subcommand() {
        Some(("start", _)) => start_command(),
        Some(("shutdown", _)) => shutdown_command(),
        Some(("run", _)) => run_command(),
        Some(("add", sub_matches)) => {
            let expr = sub_matches.get_one::<String>("EXPR").unwrap();
            let timeout =
sub_matches.get_one::<String>("TIMEOUT").unwrap();
                let job = sub_matches.get_one::<String>("JOB").unwrap();
                let port_option = sub_matches.get_one::<String>("port");
                add_command(expr, timeout, job, port_option);
            }
            Some(("load", sub_matches)) => {
                let file = sub_matches.get_one::<String>("file");
                load_command(file);
            }
            Some(("scheduler", sub_matches)) => {
                let action =
sub_matches.get_one::<String>("ACTION").unwrap();
                let port_option = sub_matches.get_one::<String>("port");
                scheduler_command(action, port_option);
            }
            _ => unreachable!("Exhausted list of subcommands and
subcommand_required prevents `None`"),
        }
    }

    fn start_command() {
        cronframe_folder();

        let (ip, port) = ip_and_port();
        if is_running(&ip, port) {
            println!(
                "{} address 'http://{ip}:{port}' is already busy.",
                "Error:".red().bold()
            );
            return;
        }

        let _build = Command::new("cronframe")
            .args(["run"])
            .stdin(Stdio::null())
            .stdout(Stdio::null())
            .stderr(Stdio::null())
            .spawn()
            .expect("cronframe run failed");

        println!("CronFrame will soon be available at http://{ip}:{port}");
    }

    fn shutdown_command() {
        let (ip, port) = ip_and_port();
        let req_url = format!("http://{ip}:{port}/shutdown");

        match reqwest::blocking::get(req_url) {
            Ok(_) => {
                println!("CronFrame will soon shutdown.");
            }
            Err(_) => {
                println!(
```

```rust
                "{} no instance found at http://{ip}:{port}",
                "Error:".red().bold()
            );
        }
    }
}

fn run_command() {
    cronframe_folder();
    let (ip, port) = ip_and_port();
    if is_running(&ip, port) {
        println!(
            "{} address 'http://{ip}:{port}' is already busy",
            "Error:".red().bold()
        );
        return;
    }
    let _ = CronFrame::init(Some(CronFilter::CLI), true).run();
}

fn add_command(expr: &str, timeout: &str, job: &str, port_option: Option<&String>) {
    let home_dir = utils::home_dir().replace("\\", "/");

    let escaped_expr = expr.replace("/", "slh");

    let tmp: Vec<_> = if cfg!(target_os = "windows") {
        job.split("\\").collect()
    } else {
        job.split("/").collect()
    };
    let tmp = tmp.iter().filter(|x| !x.is_empty()); // needed if there is a / after the name of the create's folder
    let job_name = tmp.last().unwrap().replace(".rs", "");

    println!("Compiling {job_name} Job");

    if Path::new(&job).is_file() {
        // compile the "script" job
        let compile_command = Command::new("rustc")
            .args([
                job,
                "-o",
                &format!("{home_dir}/.cronframe/cli_jobs/{job_name}"),
            ])
            .status();

        match compile_command {
            Err(error) => {
                println!("{} {}", "Error:".red().bold(), error.to_string());
                return;
            }
```

```
                _ => (),
            }
        } else {
            // compile the "crate" job
            let compile_command = Command::new("cargo")
                .args([
                    "build",
                    "--release",
                    "--target-dir",
                    &format!("{home_dir}/.cronframe/cargo_targets/{job_name}"),
                ])
                .current_dir(job)
                .status();

            match compile_command {
                Err(error) => {
                    println!("{} {}", "Error:".red().bold(), error.to_string());
                    return;
                }
                _ => (),
            }

            let copy_command = if cfg!(target_os = "windows") {
                println!(
                    "current dir = {}",
                    format!("{home_dir}/.cronframe/cargo_targets/{job_name}/release")
                );
                println!(
                    "cmd /C copy {} {}",
                    format!("{job_name}.exe"),
                    format!("{home_dir}/.cronframe/cli_jobs").replace("\\", "/")
                );

                Command::new("cmd")
                    .args(&[
                        "/C",
                        "copy",
                        &format!("{job_name}.exe"),
                        &format!("{home_dir}/.cronframe/cli_jobs/").replace("/", "\\"),
                    ])
                    .current_dir(format!(
                        "{home_dir}/.cronframe/cargo_targets/{job_name}/release"
                    ))
                    .status()
            } else {
                // copy binary on unix systems
                Command::new("cp")
```

```rust
                .args([
                    &job_name,
                    &format!("{home_dir}/.cronframe/cli_jobs/{job_name}"),
                ])
                .current_dir(format!(
                    "{home_dir}/.cronframe/cargo_targets/{job_name}/release"
                ))
                .status()
        };

        match copy_command {
            Err(error) => {
                println!("{} {}", "Error:".red().bold(), error.to_string());
                return;
            }
            _ => (),
        }
    }

    // get the ip_address and port
    // check if a cronframe instance is running
    // send the job to the running cronframe instance
    // localhost::8098/add_cli_job/<expr>/<timeout>/<job>

    let (ip, mut port) = ip_and_port();

    if port_option.is_some() {
        port = port_option.unwrap().parse().unwrap();
    }

    if !is_running(&ip, port) {
        println!(
            "{} no instance found at http://{ip}:{port}",
            "Error:".red().bold()
        );
        return;
    }

    let req_url = format!("http://{ip}:{port}/add_cli_job/{escaped_expr}/{timeout}/{job_name}");

    match reqwest::blocking::get(req_url) {
        Ok(_) => {
            println!("Added Job to CronFrame");
            println!("  Name: {job_name}");
            println!("  Cron Expression: {expr}");
            println!("  Timeout: {timeout}");
        }
        Err(error) => {
            println!("{} {error}", "Error:".red().bold());
        }
    }
}

fn scheduler_command(action: &str, port_option: Option<&String>) {
    let (ip, mut port) = ip_and_port();

    if port_option.is_some() {
        port = port_option.unwrap().parse().unwrap();
    }

    if !is_running(&ip, port) {
        println!(
            "{} no instance found at http://{ip}:{port}",
            "Error:".red().bold()
        );
        return;
    }

    match action.to_lowercase().as_str() {
        "start" => {
            let req_url = format!("http://{ip}:{port}/start_scheduler");

            match reqwest::blocking::get(req_url) {
                Ok(_) => {
                    println!("Scheduler will soon start.");
                }
                Err(error) => {
                    println!("Error when starting the scheduler");
                    println!("{error}");
                }
            }
        }
        "stop" => {
            let req_url = format!("http://{ip}:{port}/stop_scheduler");

            match reqwest::blocking::get(req_url) {
                Ok(_) => {
                    println!("Scheduler will soon stop.");
                }
                Err(error) => {
                    println!(
                        "{} {error} when stopping the scheduler",
                        "Error:".red().bold()
                    );
                }
            }
        }
        other => {
            println!("{} '{other}' action unknown.",
"Error:".red().bold());
        }
    }
```

```
}

fn load_command(file: Option<&String>) {
    let (ip, port) = ip_and_port();
    if !is_running(&ip, port) {
        println!(
            "{} no instance found at http://{ip}:{port}",
            "Error:".red().bold()
        );
        return;
    }

    let file_path = match file {
        Some(path) => path.clone(),
        None => format!("{}/.cronframe/job_list.txt", utils::home_dir()),
    };

    match std::fs::read(file_path) {
        Ok(content) => {
            for line in content.lines().into_iter() {
                let line = line.unwrap();
                let cmpt: Vec<_> = line.split(" ").collect();

                let expr = if cmpt.len() == 9 {
                    // expr made of 7 fields
                    format!(
                        "{} {} {} {} {} {} {}",
                        cmpt[0], cmpt[1], cmpt[2], cmpt[3], cmpt[4], cmpt[5], cmpt[6]
                    )
                } else {
                    // expr made of 6 fields (year absent)
                    format!(
                        "{} {} {} {} {} {}",
                        cmpt[0], cmpt[1], cmpt[2], cmpt[3], cmpt[4], cmpt[5]
                    )
                };

                let timeout = if cmpt.len() == 9 { cmpt[7] } else { cmpt[6] };
                let job = if cmpt.len() == 9 { cmpt[8] } else { cmpt[7] };

                add_command(&expr, timeout, job, None);
            }
        }
        Err(err) => {
            println!("{}", err.to_string());
        }
    }
}
```

```rust

fn cronframe_folder() {
    let home_dir = utils::home_dir();

    if !std::path::Path::new(&format!("{home_dir}/.cronframe")).exists() {
        println!("Generating .cronframe directory content...");

        let template_dir = format!("{home_dir}/.cronframe/templates").replace("\\", "/");
        let rocket_toml = format!("[debug]\ntemplate_dir = \"{template_dir}\"\n[release]\ntemplate_dir = \"{template_dir}\"");

        fs::create_dir(format!("{home_dir}/.cronframe"))
            .expect("could not create .cronframe directory");
        fs::create_dir(format!("{home_dir}/.cronframe/cli_jobs"))
            .expect("could not create .cronframe directory");

        web_server::generate_template_dir();

        let _ = fs::write(
            Path::new(&format!("{home_dir}/.cronframe/rocket.toml")),
            rocket_toml,
        );
    }
}

fn is_running(ip: &str, port: u16) -> bool {
    match reqwest::blocking::get(format!("http://{ip}:{port}")) {
        Ok(_) => true,
        Err(_) => false,
    }
}
```

B - Code Listing for cronframe_macro crate

B-1 - src/lib.rs

```rust
//! Macros for [CronFrame](https://crates.io/crates/cronframe)

use proc_macro::*;
use quote::{format_ident, quote, ToTokens};
use syn::{self, parse_macro_input, punctuated::Punctuated, ItemFn, ItemImpl, ItemStruct, Meta};

/// Global Job definition Macro
#[proc_macro_attribute]
pub fn cron(att: TokenStream, code: TokenStream) -> TokenStream {
    let args = parse_macro_input!(att with Punctuated::<Meta, syn::Token![,]>::parse_terminated);

    let args = args.into_iter().map(|x| {
        x.require_name_value()
            .map(|x| {
                let arg_name = x.path.to_token_stream().to_string();
                let arg_val = x.value.to_token_stream().to_string();
                (arg_name, arg_val.replace("\"", ""))
            })
            .unwrap()
    });

    // should contain ("expr", "* * * * * *")
    let (arg_1_name, cron_expr) = args.clone().peekable().nth(0).unwrap();

    // should contain ("timeout", "u64")
    let (arg_2_name, timeout) = args.peekable().nth(1).unwrap();

    if arg_1_name == "expr" && arg_2_name == "timeout" {
        let parsed = syn::parse::<ItemFn>(code.clone());

        if parsed.is_ok() {
            let origin_function = parsed.clone().unwrap().to_token_stream();
            let ident = parsed.clone().unwrap().sig.ident;
            let job_name = ident.to_string();

            let new_code = quote! {
                // original function
                #origin_function

                // necessary for automatic job collection
                cronframe::submit! {
                    cronframe::JobBuilder::global_job(#job_name, #ident, #cron_expr, #timeout)
                }
            };

            return new_code.into();
        } else if let Some(error) = parsed.err() {
            println!("parse Error: {}", error);
        }
    }
    code
}

/// Cron Object definition Macro
#[proc_macro_attribute]
pub fn cron_obj(_att: TokenStream, code: TokenStream) -> TokenStream {
    let item_struct = syn::parse::<ItemStruct>(code.clone()).unwrap();
    let r#struct = item_struct.to_token_stream();
    let ident_upper = format_ident!("{}", item_struct.ident.clone().to_string().to_uppercase());
    let struct_name = item_struct.ident;
    let method_jobs = format_ident!("CRONFRAME_METHOD_JOBS_{}",
ident_upper);
    let function_jobs = format_ident!("CRONFRAME_FUNCTION_JOBS_{}",
ident_upper);
    let cf_fn_jobs_flag = format_ident!("CF_FN_JOBS_FLAG_{}",
ident_upper);
    let cf_fn_jobs_channels = format_ident!("CF_FN_JOBS_CHANNELS_{}",
ident_upper);

    // inject the tx field for the drop of method jobs
    // this requires that the last field in the original struct is
followed by a ,
    let struct_edited: proc_macro2::TokenStream = {
        let mut tmp = r#struct.to_string();
        if tmp.contains("{") {
            tmp.insert_str(
                tmp.chars().count() - 1,
                "tx: Option<cronframe::Sender<String>>",
            );
        } else {
            tmp.insert_str(
                tmp.chars().count() - 1,
                "{tx: Option<cronframe::Sender<String>>}",
            );
            tmp = (&tmp[0..tmp.len() - 1].to_string()).clone();
        }
        tmp.parse().unwrap()
    };

    // --- start --- building the new_cron_obj function
    let new_cron_obj: proc_macro2::TokenStream = {
        let type_name =
struct_name.clone().into_token_stream().to_string();
        let mut function = String::from("fn new_cron_obj(");
        if !item_struct.fields.is_empty() {
            let mut tmp = item_struct.fields.iter().map(|x| {
                let field_name = x.ident.to_token_stream().to_string();
                let field_type = x.ty.to_token_stream().to_string();
                format!("{field_name} : {field_type},")
            });

            for _ in 0..item_struct.fields.len() {
                function.push_str(&tmp.next().unwrap());
            }
        }

        function.push_str(") -> ");
        function.push_str(&type_name);
        function.push_str("{");
        function.push_str(&type_name);
        function.push_str("{");

        if !item_struct.fields.is_empty() {
            let mut tmp = item_struct.fields.iter().map(|x| {
                let field_name = x.ident.to_token_stream().to_string();
                format!("{field_name},")
            });

            for _ in 0..item_struct.fields.len() {
                function.push_str(&tmp.next().unwrap());
            }
        }
        function.push_str("tx: None");
        function.push_str("}");
        function.push_str("}");
        function.parse().unwrap()
    }; // --- end --- building the new_cron_obj

    let new_code = quote! {
        // the code of the original struct with the addition of the tx field
        #[derive(Clone)]
        #struct_edited

        // used to keep track of weather function jobs have been gathered
        static #cf_fn_jobs_flag: std::sync::Mutex<bool> = std::sync::Mutex::new(false);
        // channels used to manage to drop of function jobs
        static #cf_fn_jobs_channels: cronframe::Lazy<(cronframe::Sender<String>, cronframe::Receiver<String>)> = cronframe::Lazy::new(|| cronframe::bounded(1));

        // drop for method jobs
        impl Drop for #struct_name {
            // this drops method jobs only
            fn drop(&mut self) {
                if self.tx.is_some(){
                    let _= self.tx.as_ref().unwrap().send("JOB_DROP".to_string());
                }
            }
        }

        // drop for function jobs
        impl #struct_name {
            // the new_cron_obj function
            #new_cron_obj

            // associated funciton of cron objects to drop function jobs
            fn cf_drop_fn() {
                if *#cf_fn_jobs_flag.lock().unwrap(){
                    for func in #function_jobs{
                        let _= #cf_fn_jobs_channels.0.send("JOB_DROP".to_string());
                    }
                    *#cf_fn_jobs_flag.lock().unwrap() = false;
```

```rust
                }
            }
        }

        #[cronframe::distributed_slice]
        static #method_jobs: [fn(std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>) -> cronframe::JobBuilder<'static>];

        #[cronframe::distributed_slice]
        static #function_jobs: [fn() -> cronframe::JobBuilder<'static>];
    };

    new_code.into()
}

/// Cron Implementation Block Macro
#[proc_macro_attribute]
pub fn cron_impl(_att: TokenStream, code: TokenStream) -> TokenStream {
    let item_impl = syn::parse::<ItemImpl>(code.clone()).unwrap();
    let r#impl = item_impl.to_token_stream();
    let impl_items = item_impl.items.clone();
    let impl_type = item_impl.self_ty.to_token_stream();

    let impl_type_upper = format_ident!(
        "{}",
        item_impl
            .self_ty
            .to_token_stream()
            .to_string()
            .to_uppercase()
    );

    let method_jobs = format_ident!("CRONFRAME_METHOD_JOBS_{impl_type_upper}");
    let function_jobs = format_ident!("CRONFRAME_FUNCTION_JOBS_{impl_type_upper}");

    let mut new_code = quote! {
        #r#impl
    };

    let mut count = 0;
    for item in impl_items {
        let item_token = item.to_token_stream();
        let item_fn_parsed = syn::parse::<ItemFn>(item_token.into());
        let item_fn_id = item_fn_parsed.clone().unwrap().sig.ident;
        let helper = format_ident!("cron_helper_{}", item_fn_id);
        let item_fn_id_upper = format_ident!(
            "{}",
            item_fn_id.to_token_stream().to_string().to_uppercase()
        );
        let linkme_deserialize = format_ident!("LINKME_{}_{count}", item_fn_id_upper);

        let new_code_tmp = if check_self(&item_fn_parsed) {
            // method job
            quote! {
                #[cronframe::distributed_slice(#method_jobs)]
                static #linkme_deserialize: fn(_self: std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>)-> cronframe::JobBuilder<'static> = #impl_type::#helper;
            }
        } else {
            // function job
            quote! {
                #[cronframe::distributed_slice(#function_jobs)]
                static #linkme_deserialize: fn()-> cronframe::JobBuilder<'static> = #impl_type::#helper;
            }
        };

        new_code.extend(new_code_tmp.into_iter());
        count += 1;
    }

    let type_name = impl_type.to_string().to_uppercase();

    let cf_fn_jobs_flag = format_ident!("CF_FN_JOBS_FLAG_{}", type_name);
    let cf_fn_jobs_channels = format_ident!("CF_FN_JOBS_CHANNELS_{}", type_name);

    let gather_fn = quote! {
        impl #impl_type{
            pub fn cf_gather_mt(&mut self, frame: std::sync::Arc<CronFrame>){
                cronframe::info!("Collecting Method Jobs from {}", #type_name);
                if !#method_jobs.is_empty(){
                    let life_channels = cronframe::bounded(1);
                    self.tx = Some(life_channels.0.clone());

                    for method_job in #method_jobs {
                        let job_builder = (method_job)(std::sync::Arc::new(Box::new(self.clone())));
                        let mut cron_job = job_builder.build();
                        cron_job.life_channels = Some(life_channels.clone());
                        cronframe::info!("Found Method Job \"{}\" from {}.", cron_job.name, #type_name);
                        frame.clone().add_job(cron_job);
                    }
                    cronframe::info!("Method Jobs from {} Collected.", #type_name);
                } else {
                    cronframe::info!("Not Method Jobs from {} has been
found.", #type_name);
                }
            }

            pub fn cf_gather_fn(frame: std::sync::Arc<CronFrame>){
                cronframe::info!("Collecting Function Jobs from {}", #type_name);
                if !#function_jobs.is_empty(){
                    // collect jobs from associated functions only if this is the first
                    // instance of this cron object to call the helper_gatherer function
                    let fn_flag = *#cf_fn_jobs_flag.lock().unwrap();

                    if !fn_flag {
                        for function_job in #function_jobs {
                            let job_builder = (function_job)();
                            let mut cron_job = job_builder.build();
                            cron_job.life_channels = Some(#cf_fn_jobs_channels.clone());
                            cronframe::info!("Found Function Job \"{}\" from {}.", cron_job.name, #type_name);
                            frame.clone().add_job(cron_job);
                        }
                        cronframe::info!("Function Jobs from {} Collected.", #type_name);
                        *#cf_fn_jobs_flag.lock().unwrap() = true;
                    }
                } else {
                    cronframe::info!("Not Function Jobs from {} has been found.", #type_name);
                }
            }

            pub fn cf_gather(&mut self, frame: std::sync::Arc<CronFrame>){
                self.cf_gather_mt(frame.clone());
                Self::cf_gather_fn(frame.clone());
            }
        }
    };

    new_code.extend(gather_fn.into_iter());
    new_code.into()
}

/// Function Job definition Macro for a Cron Object
#[proc_macro_attribute]
pub fn fn_job(att: TokenStream, code: TokenStream) -> TokenStream {
    let parsed = syn::parse::<ItemFn>(code.clone());

    if check_self(&parsed) {
        // self is present -> compilation error
```

```rust
    }

    // generate code for a function job
    let args = parse_macro_input!(att with Punctuated::<Meta, syn::Token![,]>::parse_terminated);

    let args = args.into_iter().map(|x| {
        x.require_name_value()
            .map(|x| {
                let arg_name = x.path.to_token_stream().to_string();
                let arg_val = x.value.to_token_stream().to_string();
                (arg_name, arg_val.replace("\"", ""))
            })
            .unwrap()
    });

    // should contain ("expr", "* * * * * *")
    let (arg_1_name, cron_expr) = args.clone().peekable().nth(0).unwrap();

    // should contain ("timeout", "time in ms")
    let (arg_2_name, timeout) = args.peekable().nth(1).unwrap();

    if arg_1_name != "expr" && arg_2_name != "timeout" {
        // wrong argument names -> compilation error
        return code;
    }

    let origin_function = parsed.clone().unwrap().to_token_stream();
    let ident = parsed.clone().unwrap().sig.ident;
    let job_name = ident.to_string();
    let helper = format_ident!("cron_helper_{}", ident);

    let new_code = quote! {
        // original function
        #[allow(dead_code)]
        #origin_function

        fn #helper() -> cronframe::JobBuilder<'static> {
            cronframe::JobBuilder::function_job(#job_name, Self::#ident, #cron_expr, #timeout)
        }
    };
    new_code.into()
}

/// Method Job definition Macro for a Cron Object
#[proc_macro_attribute]
pub fn mt_job(att: TokenStream, code: TokenStream) -> TokenStream {
    let parsed = syn::parse::<ItemFn>(code.clone());

    if !check_self(&parsed) {
        // self is missing -> compilation error
    }

    // generate code for a function job
    let args = parse_macro_input!(att with Punctuated::<Meta, syn::Token![,]>::parse_terminated);

    let args = args.into_iter().map(|x| {
        x.require_name_value()
            .map(|x| {
                let arg_name = x.path.to_token_stream().to_string();
                let arg_val = x.value.to_token_stream().to_string();
                (arg_name, arg_val.replace("\"", ""))
            })
            .unwrap()
    });

    // should contain ("expr", "name of expression field")
    let (arg_1_name, cron_expr) = args.clone().peekable().nth(0).unwrap();

    if arg_1_name != "expr" {
        // wrong argument name -> compilation error
    }

    // generate code for a method job
    let origin_method = parsed.clone().unwrap().to_token_stream();
    let ident = parsed.clone().unwrap().sig.ident;
    let job_name = ident.to_string();
    let block = parsed.clone().unwrap().block;

    let cronframe_method = format_ident!("cron_method_{}", ident);
    let helper = format_ident!("cron_helper_{}", ident);
    let expr = format_ident!("expr");
    let tout = format_ident!("tout");

    // this is to replace the native self with the self from cronframe
    let block_string = block.clone().into_token_stream().to_string();
    let mut block_string_edited = block_string.replace("self.", "cronframe_self.");
    block_string_edited.insert_str(
        1,
        "let cron_frame_instance = arg.clone();
        let cronframe_self = (*cron_frame_instance).downcast_ref::<Self>().unwrap();",
    );

    let block_edited: proc_macro2::TokenStream = block_string_edited.parse().unwrap();

    //println!("UNEDITED BLOCK:\n{block_string}");
    //println!("EDITED BLOCK:\n{block_string_edited}");

    let mut new_code = quote! {
        // original method at the user's disposal
        #[allow(dead_code)]
        #origin_method

        // cronjob method at cronframe's disposal
        // fn cron_method_<name_of_method> ...
        fn #cronframe_method(arg: std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>) #block_edited
    };

    let helper_code = quote! {
        // fn cron_helper_<name_of_method> ...
        fn #helper(arg: std::sync::Arc<Box<dyn std::any::Any + Send + Sync>>) -> cronframe::JobBuilder<'static> {
            let instance = arg.clone();
            let this_obj = (*instance).downcast_ref::<Self>().unwrap();

            let #expr = this_obj.cron_expr.expr();
            let #tout = format!("{}", this_obj.cron_expr.timeout());
            let instance = arg.clone();

            cronframe::JobBuilder::method_job(#job_name, Self::#cronframe_method, #expr.clone(), #tout, instance)
        }
    };

    // replace the placeholder cron_expr with the name of the field
    let helper_code_edited = helper_code
        .clone()
        .into_token_stream()
        .to_string()
        .replace("cron_expr", &cron_expr);
    let block_edited: proc_macro2::TokenStream = helper_code_edited.parse().unwrap();

    new_code.extend(block_edited.into_iter());

    new_code.into()
}

// aid function for fn_job and mt_job
fn check_self(parsed: &Result<ItemFn, syn::Error>) -> bool {
    if !parsed.clone().unwrap().sig.inputs.is_empty()
        && parsed
            .clone()
            .unwrap()
            .sig
            .inputs
            .first()
            .unwrap()
            .to_token_stream()
            .to_string()
            == "self"
    {
        true
    } else {
        false
    }
}
```

C - Dependencies List

1. **chrono v0.4.38**
   - https://crates.io/crates/chrono/0.4.38
2. **clap v4.5.15**
   - https://crates.io/crates/clap/4.5.15
3. **colored v2.1.0**
   - https://crates.io/crates/colored/2.1.0
4. **cron v0.12.1**
   - https://crates.io/crates/cron/0.12.1
5. **crossbeam-channel v0.5.12**
   - https://crates.io/crates/crossbeam-channel/0.5.12
6. **home v0.5.9**
   - https://crates.io/crates/home/0.5.9
7. **inventory v0.3.15**
   - https://crates.io/crates/inventory/0.3.15
8. **linkme v0.3.26**
   - https://crates.io/crates/linkme/0.3.26
9. **log v0.4.21**
   - https://crates.io/crates/log/0.4.21
10. **log4rs v1.3.0**
    - https://crates.io/crates/log4rs/1.3.0
11. **once_cell v1.19.0**
    - https://crates.io/crates/once_cell/1.19.0
12. **reqwest v0.12.5**
    - https://crates.io/crates/reqwest/0.12.5
13. **rocket v0.5.1**
    - https://crates.io/crates/rocket/0.5.1
14. **rocket_dyn_templates v0.2.0**
    - https://crates.io/crates/rocket_dyn_templates/0.2.0
15. **serde_json v1.0**
    - https://crates.io/crates/serde_json
16. **toml v0.8.14**
    - https://crates.io/crates/toml/0.8.14
17. **uuid v1.8.0**
    - https://crates.io/crates/uuid/1.8.0