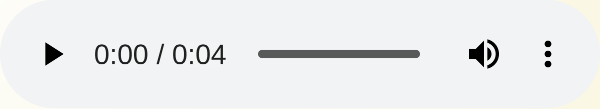
AN INITIAL COLLECTION OF

RUST CRATES

2 WORDS ABOUT ME

- Stats
  - First name: René (Uggla)
  - Last name: Ribaud
- Skills
  - Class: Software engineer
  - Previous Class: Solution architect (Cloud / Devops)
  - Latest Guilde: Red Hat
  - Game start: 1998
  - Preferred weapons: Rust / Python
  - Artefact: Openstack Nova
- Optional traits
  - Linux and FLOSS since 1995
  - Previously Ops, Dev today to produce my own bugs
  - Rust coding dojo with AlpesCraft



A QUITE SMALL STANDARD LIBRARY

- Rust's deliberate choice to keep the standard library small:
    - Ensures easier compatibility across versions.
    - Allows code to mature as external crates before potential inclusion in the standard library.
    - Promotes innovation by allowing multiple competing solutions in the ecosystem.
- As a result, you need to familiarize yourself with the Rust ecosystem to find the right crates for your needs.

ABOUT THIS SELECTION

- This selection represents **some of my favorite Rust crates**, the ones I use or have used in my projects.
- *Note:* There are often other alternatives for similar use cases. This selection therefore reflects **my personal preferences**, based on my experience and specific needs.
- This is focused on "simple" crates, excluding frameworks (e.g., Actix) and tools (e.g., Bindgen).
- Please **don’t blame me** if your favorite crate is not included in the list. We can share them afterward!
- It wasn't easy to make this selection, as there are many other interesting crates that could have been included. If you enjoy this presentation, I would be happy to showcase additional crates in the future.

HOW TO USE A CRATE? 1/3

- Add the crate name to the dependencies section of your `Cargo.toml`:

```
───────┬──────────────────────────────────────────
       │ File: Cargo.toml
───────┼──────────────────────────────────────────
       │ [package]
       │ name = "rand_example"
       │ version = "0.1.0"
       │ edition = "2021"
       │
       │ [dependencies]
       │ rand = "0.8.5"
───────┴──────────────────────────────────────────
```

- Add the namespace of the crate using the `use` statement.
- Depending on your crate:
    - You might use a **prelude** to import common public items.

```
use bevy::prelude::*;
```

    - You can import specific items explicitly if needed:

```
use rand::thread_rng;
use rand::distributions::Uniform;
```

- Cargo will download and build the crate at compile time.

HOW TO USE A CRATE? 2/3

- A crate might have features. This is useful for:
  - Compatibility, e.g., targeting specific operating systems.
  - Technology choice, e.g., selecting between OpenSSL and Rustls.
  - Reducing compilation time by enabling only the necessary functionality.

```toml
[dependencies]
rand = { version = "0.8.5", features = ["simd_support"] }
```

- A `Cargo.lock` is created to track dependencies.

HOW TO USE A CRATE? 3/3

- Before Rust 1.62, a Cargo plugin could be used to manage dependencies and features. This functionality is now part of Cargo.
- It can be installed with `cargo install cargo-edit`.
  See https://github.com/killercup/cargo-edit
- Example:

```
□ uggla □ □ main □ ~ □ workspace □ rust □ rand_example □ 1 □ cargo add rand -F simd_support
    Updating crates.io index
      Adding rand v0.8.5 to dependencies
             Features:
             + alloc
             + getrandom
             + libc
             + packed_simd
             + rand_chacha
             + simd_support
             + std
             + std_rng
             - log
             - min_const_gen
             ....
    Locking 2 packages to latest compatible versions
      Adding libm v0.1.4
      Adding packed_simd_2 v0.3.8
```

RAND

RAND 1/3

- Rand is a powerful crate for generating random numbers and performing random operations in Rust.
- It supports:
  - Pseudo-random number generation (PRNG).
  - Cryptographically secure random number generation.
  - Shuffling and sampling.
- Simple to use with:
  1. Adding the crate to your `Cargo.toml`.
  2. Using the provided utilities in your code.

RAND 2/3

```rust
use rand::Rng;

fn main() {
    let mut rng = rand::thread_rng();
    let x: u8 = rng.gen();
    println!("Random u8: {}", x);

    let y = rng.gen_range(1..101);
    println!("Random number between 1 and 100: {}", y);
}
```

RAND 3/3

```rust
use rand::seq::SliceRandom;

fn main() {
    let mut rng = rand::thread_rng();

    let mut numbers = vec![1, 2, 3, 4, 5];
    numbers.shuffle(&mut rng);
    println!("Shuffled numbers: {:?}", numbers);

    let choice = numbers.choose(&mut rng);
    println!("Random choice: {:?}", choice);
}
```

SERDE

SERDE 1/3

- Serde is a magical framework for serializing and deserializing Rust data structures into JSON, TOML, YAML, and more.
- Format supported by compagnion crates: serde_json, bincode...
- It is as simple as:
    1. Deriving a struct.
    2. Converting to and from strings.

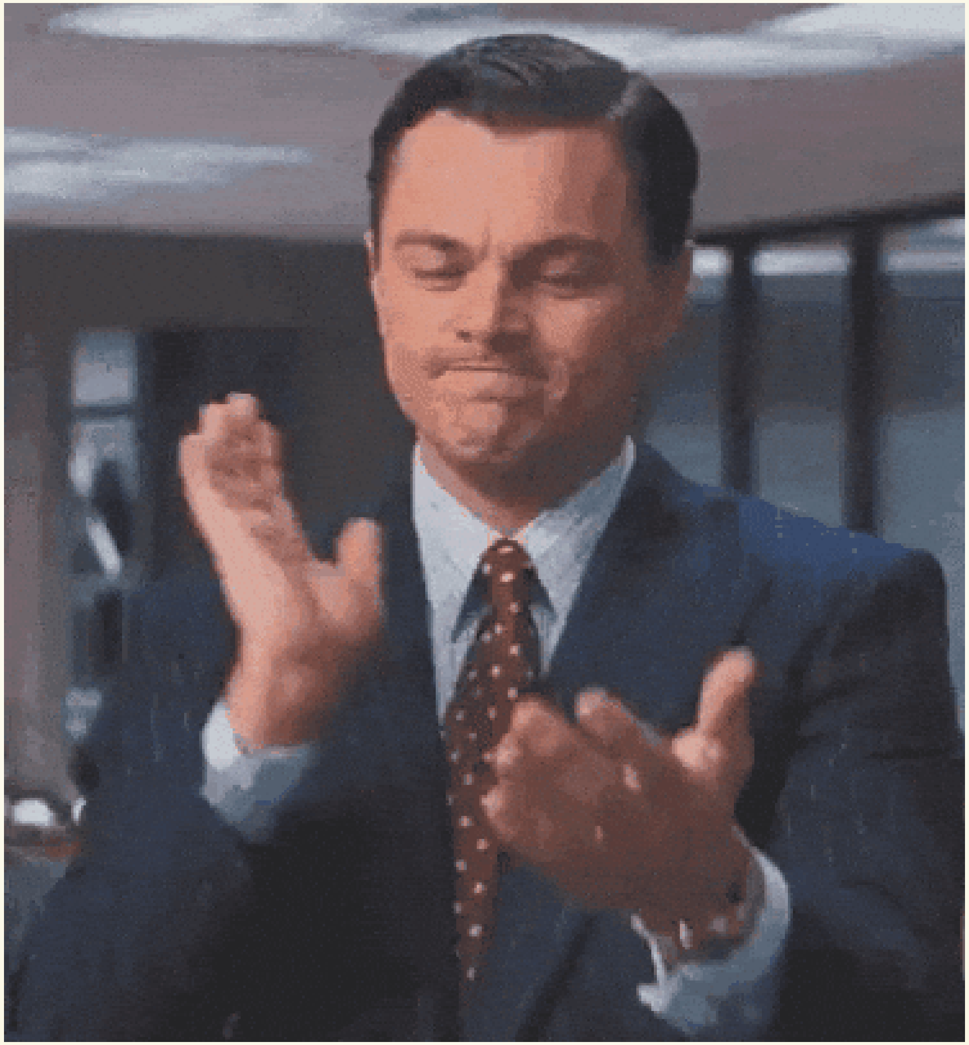
SERDE 2/3

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct User {
    id: u32,
    name: String,
    email: String,
}
```

SERDE 3/3

```rust
fn main() {
    // Sérialisation : Rust → JSON
    let user = User {
        id: 1,
        name: "Alice".to_string(),
        email: "alice@example.com".to_string(),
    };
    let json = serde_json::to_string(&user).unwrap();
    println!("JSON: {}", json);

    // Désérialisation : JSON → Rust
    let json_str = r#"
        {
            "id": 2,
            "name": "Bob",
            "email": "bob@example.com"
        }
    "#;
    let deserialized_user: User = serde_json::from_str(json_str).unwrap();
    println!("User: {:?}", deserialized_user);
}
```

CLAP



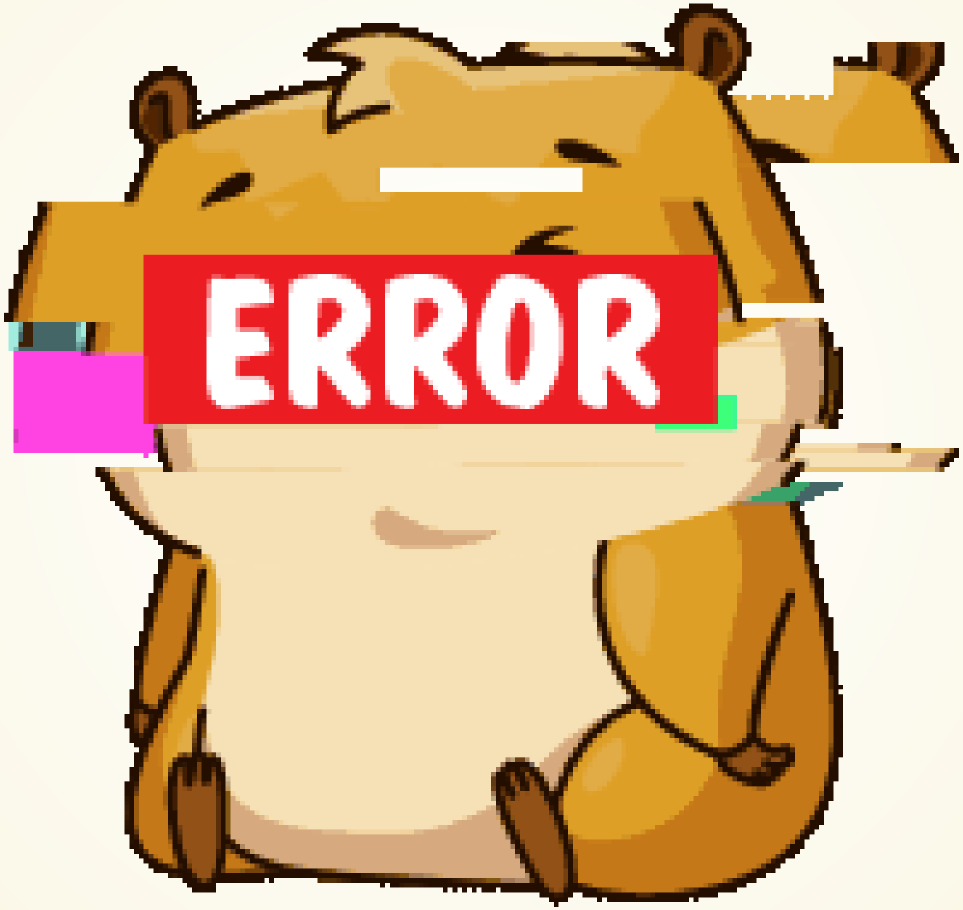
CLAP 1/2

- Already well covered in a previous meetup.
- Parse CLI arguments and options.
- Derive feature allow to define arguments and options from a struct.

CLAP 2/2

```rust
use clap::Parser;

/// Simple CLI tool example
#[derive(Parser, Debug)]
#[command(name = "greet")]
#[command(about = "A simple program to greet someone", long_about = None)]
struct Args {
    /// The name of the person to greet
    #[arg(short, long)]
    name: String,

    /// Number of times to print the greeting
    #[arg(short, long, default_value_t = 1)]
    count: u8,
}

fn main() {
    let args = Args::parse();

    for _ in 0..args.count {
        println!("Hello, {}!", args.name);
    }
}
```

THISERROR / ANYHOW



THISERROR / ANYHOW 1/2

- Already well covered in a previous meetup.
- Thiserror:
  - Provides an ergonomic way to define custom error types in Rust.
  - Automatically implements the `std::error::Error` trait for your errors.
  - Best suited for libraries where structured error types are needed.
- Anyhow
  - Simplifies error handling for applications (not libraries).
  - Offers a generic error type `anyhow::Error` to encapsulate any error.
  - Perfect for applications that don't need precise error typing.

THISERROR / ANYHOW 2/2

```rust
use anyhow::{Context};
use thiserror::Error;

#[derive(Error, Debug)]
enum MyError {
    #[error("Configuration file not found: {0}")]
    ConfigNotFound(String),

    #[error("Invalid input: {0}")]
    InvalidInput(String),
}

fn read_config(file: &str) -> Result<String, MyError> {
    if file == "missing.conf" {
        return Err(MyError::ConfigNotFound(file.to_string()).into());
    }
    if file == "invalid.conf" {
        return Err(MyError::InvalidInput("Invalid syntax".to_string()).into());
    }
    Ok("config content".to_string())
}

fn main() -> anyhow::Result<()>{
    let config =
        read_config("missing.conf").with_context(|| "Failed to load the configuration file")?;
    println!("Config: {}", config);
    Ok(())
}
```

NOM

NOM 1/2

- Nom is a parser combinators library.
- It is like regexp on steroids and more readable.
- It can work on complete strings or streams.
- The regexp library is great too, but to my mind, nom help to write more maintainable code.

NOM 2/2

```
#[derive(Debug,PartialEq)]
pub struct Color { pub red:   u8, pub green: u8, pub blue:  u8, }

fn from_hex(input: &str) -> Result<u8, std::num::ParseIntError> {
  u8::from_str_radix(input, 16)
}

fn is_hex_digit(c: char) -> bool {
  c.is_digit(16)
}

fn hex_primary(input: &str) -> IResult<&str, u8> {
  map_res(
    take_while_m_n(2, 2, is_hex_digit),
    from_hex
  )(input)
}

fn hex_color(input: &str) -> IResult<&str, Color> {
  let (input, _) = tag("#")(input)?;
  let (input, (red, green, blue)) = tuple((hex_primary, hex_primary, hex_primary))(input)?;

  Ok((input, Color { red, green, blue }))
}

#[test]
fn parse_color() {
  assert_eq!(hex_color("#2F14DF"), Ok(("", Color { red: 47, green: 20, blue: 223, })));
}
```

RAYON

RAYON 1/3

- Data parallelism library.
- Provides a lot of parallel iterators for various types.
    - Vec
    - Array
    - Ranges
    - Collections
    - ...
- Simply changing an iterator fror iter to par_iter can parallelize it.

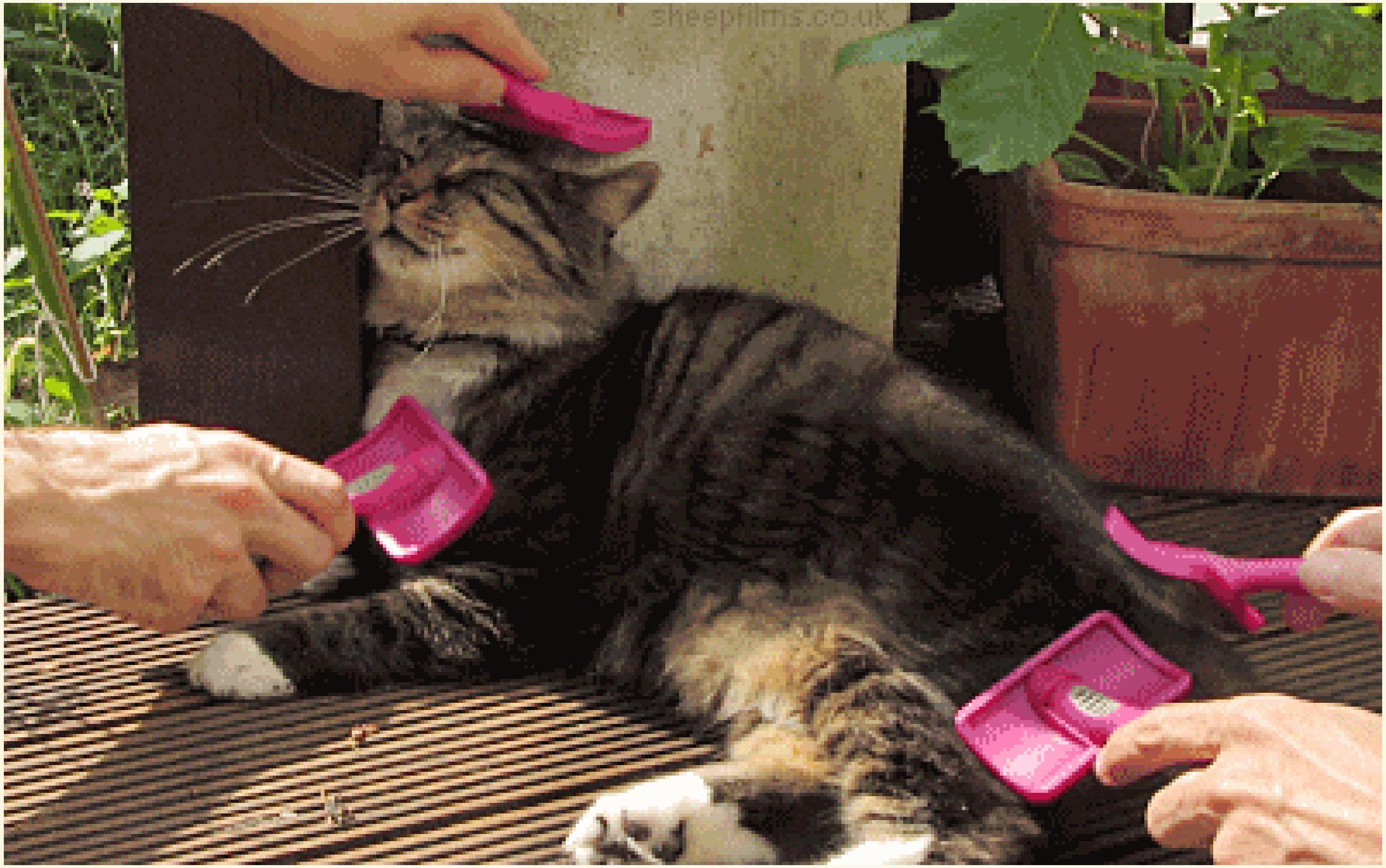
RAYON 2/3

```rust
use rayon::prelude::*;

/// Check if a number is prime
fn is_prime(n: u64) -> bool {
    if n < 2 {
        return false;
    }
    for i in 2..=((n as f64).sqrt() as u64) {
        if n % i == 0 {
            return false;
        }
    }
    true
}

/// Generate prime numbers up to a given limit
fn generate_primes(limit: u64) -> Vec<u64> {
    (2..=limit) // Create a range from 2 to the limit
        .into_par_iter() // Convert to a parallel iterator using rayon
        .filter(|&n| is_prime(n)) // Filter out non-prime numbers
        .collect() // Collect results into a vector
}

fn main() {
    let limit = 20_000_000; // Upper limit for prime numbers
    let primes = generate_primes(limit);

    println!("Found {} primes up to {}.", primes.len(), limit);
}
```

RAYON 3/3

```
 □ uggla □ □ main □ ~ □ workspace □ rust □ prime □ hyperfine target/release/prime_not_par target/release/prime
Benchmark 1: target/release/prime_not_par
  Time (mean ± σ):     13.349 s ±  0.793 s    [User: 13.276 s, System: 0.014 s]
  Range (min … max):   12.411 s … 14.910 s    10 runs

Benchmark 2: target/release/prime
  Time (mean ± σ):      3.523 s ±  0.279 s    [User: 24.565 s, System: 0.062 s]
  Range (min … max):    3.284 s …  4.236 s    10 runs

Summary
  target/release/prime ran
    3.79 ± 0.38 times faster than target/release/prime_not_par
```

ITERTOOLS



ITERTOOLS 1/2

- Itertools is a powerful crate providing additional iterator adaptors and utilities for Rust.
- Extends the standard Iterator with a wide range of combinators for advanced data processing.
- Sort, join, cartesian product, permutations, combinations, group_by, ...

ITERTOOLS 2/2

```
#[derive(Debug,PartialEq)]
use itertools::Itertools;

fn main() {
    let nums = vec![3, 2, 1];

  // Sort and join elements into a string
    let joined = nums.iter().sorted().join(",");
    assert_eq!("1,2,3", joined);

    // Generate all combinations of size 2
    let combinations: Vec<Vec<usize>> = nums.into_iter().combinations(2).collect();
    assert_eq!(vec![vec![3,2], vec![3,1],vec![2,1]], combinations);

}
```

MINREQ

MINREQ 1/2

- Simple, minimal-dependency HTTP client.
- Lightweight.
- Only sync.
- Serde integration.
- TLS support.

MINREQ 2/2

```rust
fn main() -> Result<(), minreq::Error> {
    let response = minreq::post("http://httpbin.org/anything")
        .with_body("Hello, world!")
        .send()?;

    // httpbin.org/anything returns the body in the json field "data":
    let json: serde_json::Value = response.json()?;
    assert_eq!("Hello, world!", json["data"]);

    Ok(())
}
```

ORDERED-FLOAT

ORDERED-FLOAT 1/2

- Floating-point numbers (`f32`, `f64`) cannot be directly used as keys in structures like `HashSet` or `HashMap`. This is because they do not implement necessary traits (`Eq` and `Hash`) due to special behaviors (e.g., `NaN`).
- Ordered-float provides a wrapper around floating-point types, making them orderable and usable in collections.
    - Ensures consistent comparisons, handling edge cases like `-0.0` and `0.0` as equal, and placing `NaN` consistently during sorting.
    - Values are sorted consistently, even with edge cases like `-0.0` and `NaN`.
    - `HashSet` ignores duplicates and treats `-0.0` and `0.0` as identical.
- Solves the problem where standard floating-point numbers cannot be used in such structures.

ORDERED-FLOAT 2/2

```rust
use ordered_float::OrderedFloat;
use std::collections::HashSet;

fn main() {
    // Exemple 1 : Tri de nombres flottants
    let mut floats = vec![3.2, 1.5, 2.8, 4.1, -0.0, 0.0, f64::NAN];
    //floats.sort_by(|a,b| a.partial_cmp(b).unwrap_or(std::cmp::Ordering::Equal));
    floats.sort_by_key(|&x| OrderedFloat(x)); // Tri avec OrderedFloat

    println!("Sorted floats: {:?}", floats);

    // Exemple 2 : Utilisation dans un HashSet
    let mut set: HashSet<OrderedFloat<f64>> = HashSet::new();
    set.insert(OrderedFloat(3.2));
    set.insert(OrderedFloat(1.5));
    set.insert(OrderedFloat(1.5)); // Duplicate, ne sera pas ajouté
    set.insert(OrderedFloat(-0.0)); // -0.0 et 0.0 sont considérés égaux
    set.insert(OrderedFloat(0.0));

    println!("HashSet contains: {:?}", set);
}
```

INDICATIF

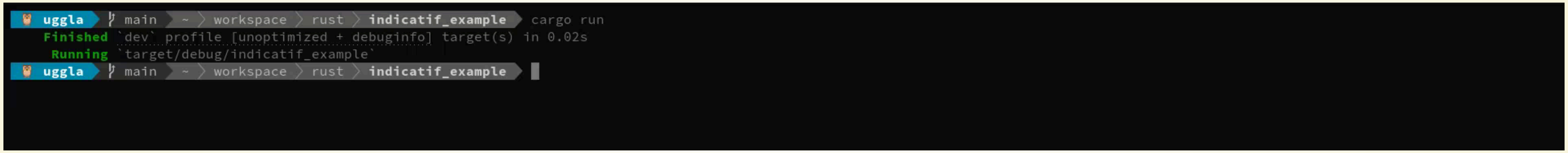
INDICATIF 1/3

- Allow to create progress bars.
- It comes with various tools and utilities for formatting anything that indicates progress.
- Similar to tqdm in python.

INDICATIF 2/3

```rust
use std::thread;
use std::time::Duration;

use indicatif::{ProgressBar, ProgressIterator, ProgressStyle};

fn main() {
    // Default styling, attempt to use Iterator::size_hint to count input size
    for _ in (0..1000).progress() {
        // ...
        thread::sleep(Duration::from_millis(5));
    }
    // Provide explicit number of elements in iterator
    for _ in (0..1000).progress_count(1000) {
        // ...
        thread::sleep(Duration::from_millis(5));
    }
    // Provide a custom bar style
    let pb = ProgressBar::new(1000);
    pb.set_style(
        ProgressStyle::with_template(
            "{spinner:.green} [{elapsed_precise}] [{bar:40.cyan/blue}] ({pos}/{len}, ETA {eta})",
        )
        .unwrap(),
    );
    for _ in (0..1000).progress_with(pb) {
        // ...
        thread::sleep(Duration::from_millis(5));
    }
}
```

INDICATIF 3/3



```
uggla  main  ~  workspace  rust  indicatif_example  cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/indicatif_example`
uggla  main  ~  workspace  rust  indicatif_example
```

LOG / SIMPLELOGGER

LOG / SIMPLELOGGER 1/2

- Just a logger to display formatted logs to stdout.
- Extremely simple to use, based on the log crate.
- However today you might use the tracing crate as a replacement.

LOG / SIMPLELOGGER 2/2

```
use simple_logger::SimpleLogger;

fn main() {
    SimpleLogger::new().init().unwrap();

    log::warn!("This is an example message.");
}
```

```
2024-01-19T17:37:07.013874956Z WARN [logging_example] This is an example message.
```

RSTEST

RSTEST 1/2

- A library to extend tests features.
- Add fixture and parametric tests.
- Reduce the number of tests to write without using macros.

RSTEST 2/2

```rust
use rstest::rstest;

#[rstest]
#[case(0, 0)]
#[case(1, 1)]
#[case(2, 1)]
#[case(3, 2)]
#[case(4, 3)]
#[case(5, 5)]
#[case(6, 8)]
fn fibonacci_test(#[case] input: u32,#[case] expected: u32) {
    assert_eq!(expected, fibonacci(input))
}

fn fibonacci(input: u32) -> u32 {
    match input {
        0 => 0,
        1 => 1,
        n => fibonacci(n - 2) + fibonacci(n - 1)
    }
}
```

IMAGE

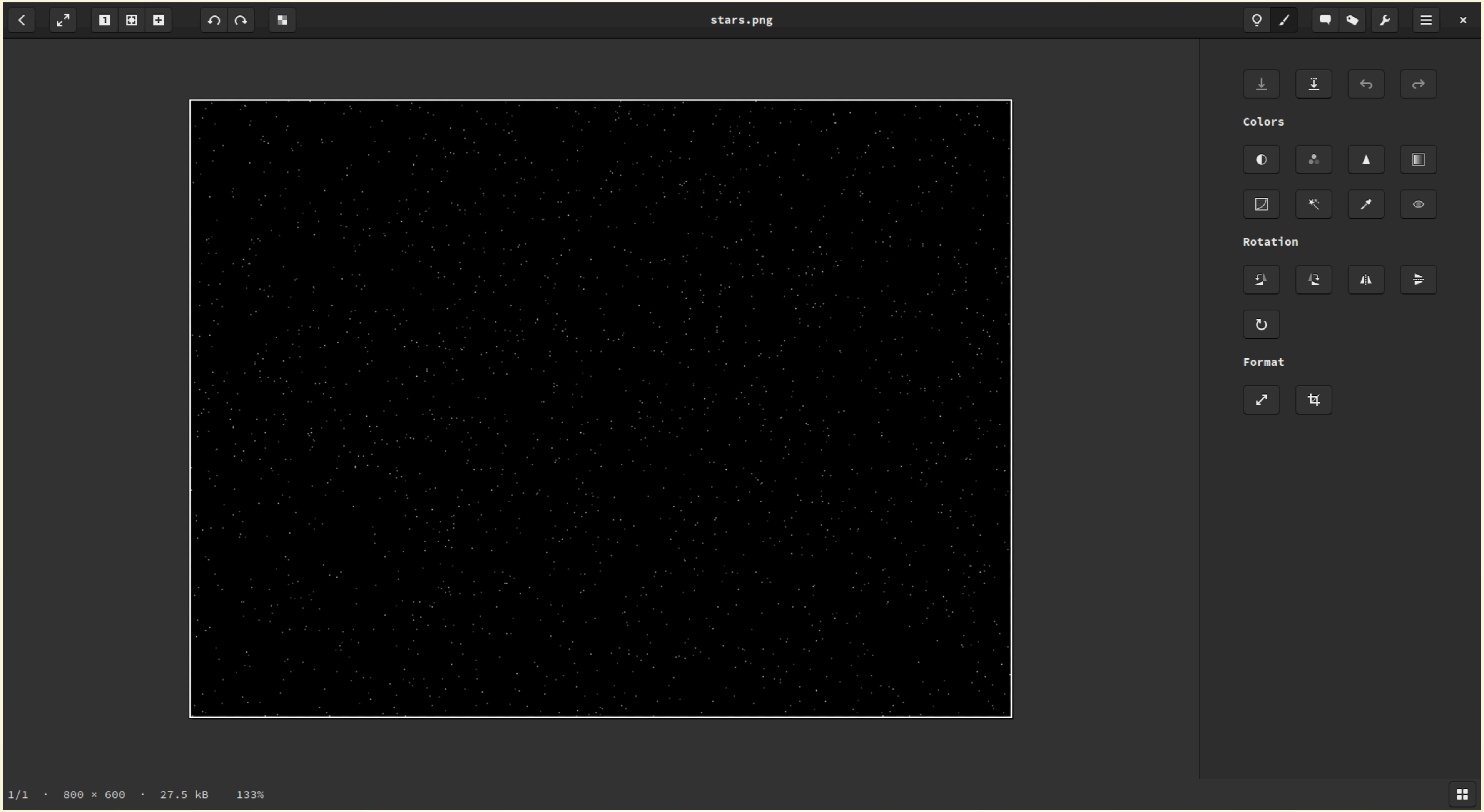
IMAGE 1/3

- Image processing library.
- Support a lot of formats (png, jpeg, bmp...).
- Can read, write and basically manipulate images.

IMAGE 2/3

```rust
const IMG_WIDTH: u32 = 800;
const IMG_HEIGHT: u32 = 600;

fn main() {
    // Image dimensions
    let width = IMG_WIDTH;
    let height = IMG_HEIGHT;

    // Create image
    let mut img = image::RgbImage::new(width, height);

    // Number of stars
    let num_stars = 2000;

    // Random number generator
    let mut rng = rand::thread_rng();

    for _ in 0..num_stars {
        let channel_color = rand::Rng::gen_range(&mut rng, 100..255);

        let x = rand::Rng::gen_range(&mut rng, 0..width);
        let y = rand::Rng::gen_range(&mut rng, 0..height);

        img.put_pixel(
            x,
            y,
            image::Rgb([channel_color, channel_color, channel_color]),
        );
    }
    img.save("stars.png").unwrap();
}
```

IMAGE 3/3



RAQOTE

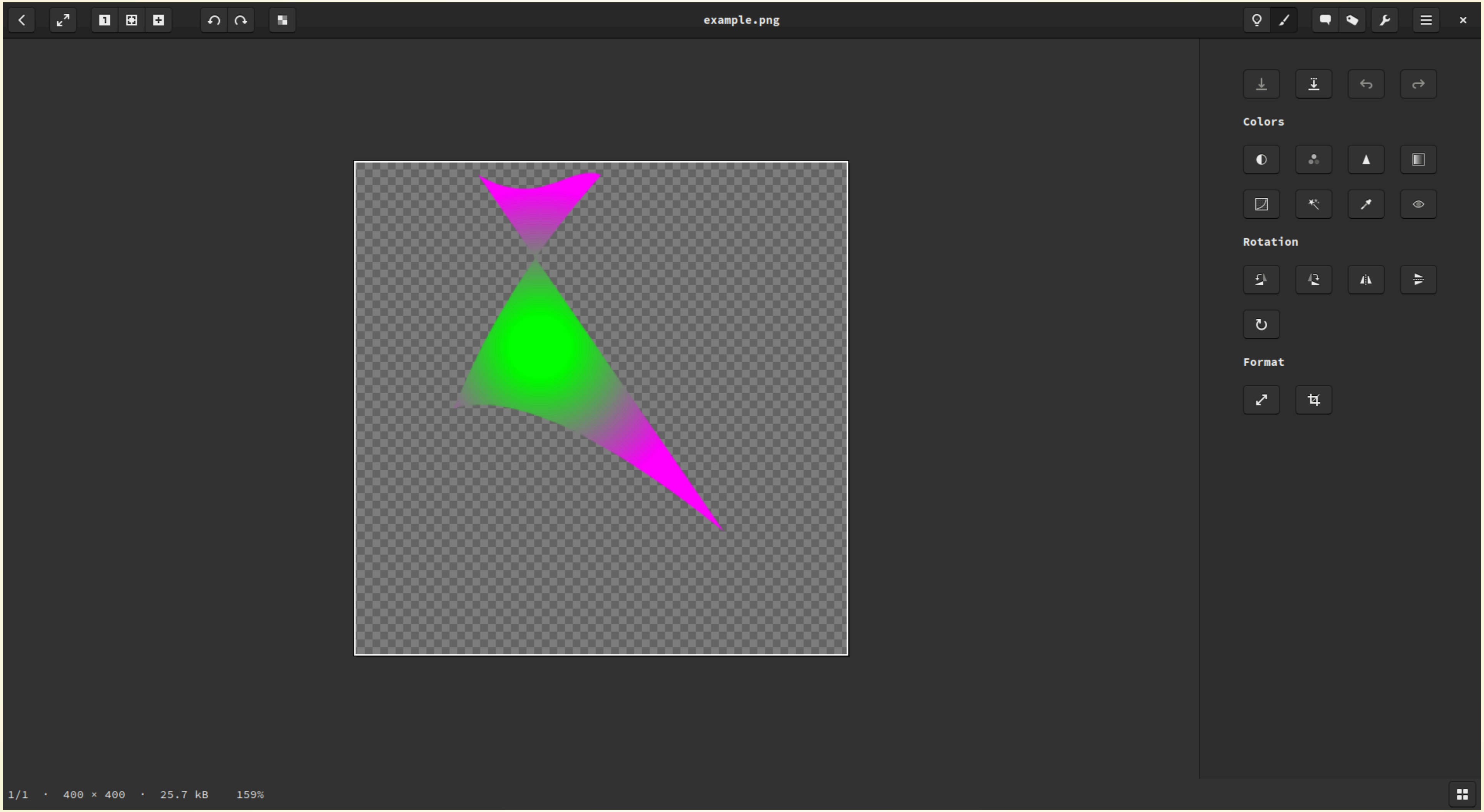
RAQOTE 1/3

- Software 2D graphics library.
- Used by servo as canvas backend.
- Features:
    - path filling
    - stroking
    - dashing
    - image, solid, and gradient fills
    - rectangular and path clipping
    - blend modes
    - layers
    - repeat modes for images
    - global alpha

RAQOTE 2/3

```rust
use raqote::*;
fn main() {
    let mut dt = DrawTarget::new(400, 400);
    let mut pb = PathBuilder::new();
    pb.move_to(100., 10.);
    pb.cubic_to(150., 40., 175., 0., 200., 10.);
    pb.quad_to(120., 100., 80., 200.);
    pb.quad_to(150., 180., 300., 300.);
    pb.close();
    let path = pb.finish();
    let gradient = Source::new_radial_gradient(
        Gradient {
            stops: vec![
                GradientStop {
                    position: 0.2,
                    color: Color::new(0xff, 0, 0xff, 0),
                },
                GradientStop {
                    position: 1.,
                    color: Color::new(0xff, 0xff, 0, 0xff),
                },
            ],
        },
        Point::new(150., 150.), 128., Spread::Pad,
    );
    dt.fill(&path, &gradient, &DrawOptions::new());
    let _ = dt.write_png("example.png");
}
```

RAQOTE 3/3

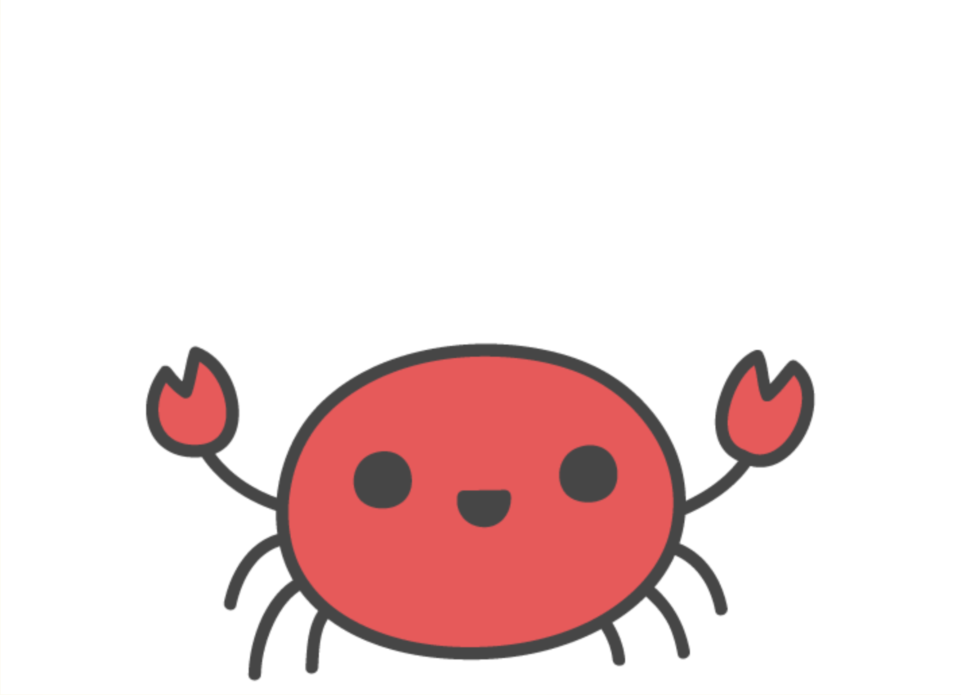


BLESSED

HTTPS://BLESSED.RS/CRATES

WHAT ARE YOU USING ?

THANKS



- René Ribaud <rene.ribaud@gmail.com>