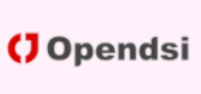
ASYNC RUST AND CLI

ALPOSS 2026

Nos sponsors Premium

UGA
Université
Grenoble Alpes

Nos sponsors



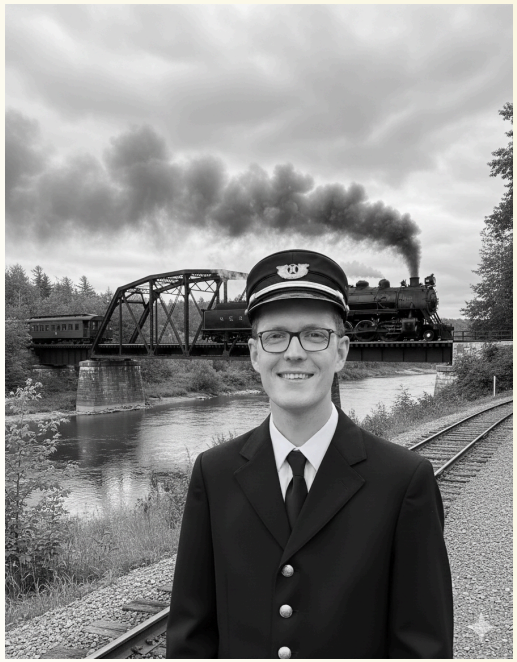
ASYNC RUST AND CLI

WELCOME AND DEVELOPMENT ENVIRONMENT SETUP

- Save time by doing the setup if not already done or as an alternative use https://github.com/codespaces.
    1. **Install Rust**: rustup, rustc, cargo, clippy, rustfmt.
    2. **Configure Editor**: Set up your preferred editor with Rust Analyzer.
    3. **Clone the workshop repository** from GitHub. https://github.com/uggla/async_rust_cli
    4. Get your personal SNCF API key https://numerique.sncf.com/startup/api/token-developpeur/
        - If you don’t want to create a personal API key for the workshop, you can download a pre-generated one instead (*workshop use only*).
        - Download link: `http://<we_will_give_the_ip>>:8000/apikey.html`
- Then we will do the introduction.

A FEW WORDS ABOUT US 1/2

- Stats
  - First name: Cyril
  - Last name: Marin
- Skills
  - Class: Cloud engineer
  - Latest Guild: Mimosa
  - Age of Experience: 10+ years
  - Preferred weapons: YAML, Shell
- Optional traits
  - Lone wolf
  - Ex-developer but still coder
  - The only tuxedo user among a whole team of half bitten apple (Does that make me a rebel undercover?)



A FEW WORDS ABOUT US 2/2

- Stats
  - First name: René (Uggla)
  - Last name: Ribaud
- Skills
  - Class: Software engineer
  - Previous Class: Solution architect (Cloud / Devops)
  - Latest Guild: 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 bugs

DISCLAIMER & QUICK POLL

- This session is labeled as **intermediate**.
  - If you missed it, no worries — it happens quite often
- To adapt the pace and the level of details, let’s do a quick poll:
  - What is your Rust experience?
    - new to Rust / beginner
    - some Rust experience
    - comfortable with Rust
  - What is your main background?
    - low-level (C / C++)
    - systems / backend (Rust, Go, Java, etc.)
    - high-level (Python, JavaScript, etc.)
- Based on this:
  - we may go through some **optional slides**,
  - or skip them if everyone is already comfortable.

QUICK OVERVIEW OF RUST (OPTIONAL)

- Provide an alternative to C/C++ and also higher-level languages
- Multi paradigm language (imperative, functional, object oriented (not fully))
- Fast, safe, and efficient (modern)
- No garbage collector, ownership and borrow checker
- Dual license MIT and Apache v2.0
- First stable release May 15th, 2015

COMMON MISCONCEPTION (OPTIONAL)

RUST HAS A HARD LEARNING CURVE

- **Yes**: Rust offers many powerful features and concepts, which can make the language look difficult at first. However, you can write **simple and idiomatic Rust** without using everything, and learn the language **step by step**.
- But many frustrations come from a different root cause:
    - People **don’t read the book** (at least the key chapters),
    - They try to **transpose habits from other languages**, and quickly hit Rust’s unique concepts (ownership/borrowing, lifetimes, traits…).
    - Developers coming from **C/C++** sometimes feel **limited or constrained by the compiler**.
    - People often **don’t read the compiler error messages**, even though they are usually precise and actionable.

SOME OF THE MAIN DIFFICULTIES (OPTIONAL)

- **1) Immutability**
  - **Immutable by default** (you opt-in with `mut`).
  - Immutable ≠ constant:
    - **const** / compile-time constants
    - **immutable bindings** / runtime values that simply cannot be modified
- **2) Ownership** (because there is no GC)
  - Goal: **prevent memory errors** (use-after-free, double free, data races).
  - Core tool: **borrowing** (use references to avoid moving ownership).
- **3) Type system**
  - An extremely strict system built on **strong typing**.
  - It pushes you to make states explicit (enums), model invariants, and handle errors properly.
  - **No nulls**: there is no `null`. Instead, Rust encodes absence and errors in the type system:
    - `Option<T>` for “value or nothing”
    - `Result<T, E>` for “success or error”

    This can feel **confusing at first**, but it forces you to handle all cases explicitly.

QUICK OVERVIEW OF THE PROJECT

- Goals of the Workshop:
  - Structure a **small and near-realistic Rust project**
  - Use key crates from the ecosystem: **Tokio**, **Reqwest**, **Serde**, **Clap**, **Anyhow**, etc.
  - Consume a real-time API (e.g. SNCF data)
  - Handle asynchronism to refresh data **without blocking the UI**
  - Showcase many **Rust language features** (ownership, borrowing, async/await, traits, error handling, etc.)
- What we will build together:
  - A CLI dashboard for train departures between two stations

QUICK OVERVIEW OF THE SESSION

- Each step starts with **slides**:
  - explanation of the goal,
  - key concepts and points of attention.
  - the full project is around **~2000 lines of code**, which is not something we can build from scratch during a workshop
  - this explains the approach: we progressively explore and evolve an existing codebase
- Then we move to the code:
  - we use `cargo t` to run the test suite,
  - the goal is usually to **fix the code**, with errors **intentionally focused** on the topic of the current part
  - but fixing tests is **not the main objective**
  - the real goal is to:
    - understand the code,
    - identify patterns and design choices,
    - and ask questions when something is unclear.
  - About using **LLMs**:
    - using an LLM to **understand or explain the code** is totally fine,
    - but using it to **directly fix the code** is a bit unfortunate,
    - the value of the workshop comes from **reasoning, discussing, and experimenting together**.
- We decide **together** how to proceed:
  - we fix things live if needed,
  - or we move on if the audience feels comfortable.

PART 1 - SETUP AND TRACING

PART 1 – SETUP YOUR PROJECT CORRECTLY

- **Workspace-based project**
  - The project is organized as a **Cargo workspace**.
  - It allows us to split responsibilities into multiple crates and **improve compilation times**.
- **API crate**
  - A dedicated crate is used to interact with the external API.
  - Benefits:
    - clear separation of concerns,
    - reusable logic,
    - faster rebuilds when working on the UI.
- **Crate boundaries**
  - `lib.rs` — public API of a crate
  - `main.rs` — binary entry point
  - The binary imports and uses the API crate like any external dependency.

PART 1 – SETUP YOUR PROJECT CORRECTLY (CONT.)

- **Visibility & API design**
  - Items are **private by default**.
  - `pub` defines what is exposed outside the crate.
  - Crate boundaries act as a **clear API contract**.
- **Error handling strategy**
  - `thiserror` — structured errors in **libraries**
  - `anyhow` — ergonomic error handling in **binaries**
  - Rule of thumb:
    - **Libraries**: typed, explicit errors
    - **Binaries**: flexible error propagation

PART 1 – CRATES

- **dotenvy**
  - Loads environment variables from a `.env` file.
  - Useful to keep configuration outside the code (API keys, endpoints, feature flags).
- **tracing**
  - The **go-to crate** to instrument Rust applications.
  - Designed to measure and observe execution:
    - structured logs,
    - timings and latencies,
    - **spans** to follow execution paths.
  - Can export data to **OpenTelemetry** for advanced observability.
  - In this workshop:
    - we rely on **`log` compatibility**,
    - and write logs to a **file**,
    - which is essential for debugging because **stdout / stderr are not available in CLI mode**.

PART 2 - ASYNC AND MPSC

PART 2 – GOALS

- We move to **async** early on:
  - to avoid blocking the future TUI loop,
  - to later perform **real API requests asynchronously**,
  - without changing the overall architecture.
- For now, the **CLI loop is simulated** inside the `run()` function.
  - It represents the future event loop: rendering, input handling, and state updates.
- A **background task** simulates periodic data updates:
  - it acts like the SNCF API refresh,
  - sends new data at regular intervals,
  - communicates with the CLI loop via a channel.

PART 2 – ASYNC

- **You need an executor** to run async code
  - `async fn` returns a **Future** (it does not run by itself)
  - An executor is responsible for **polling** futures and making progress
  - **Tokio** is the most common one, but it is **not the only executor** (for example **Embassy** for embedded systems)
- **Tokio API looks similar to std threads**
  - `tokio::spawn(...)` vs `std::thread::spawn(...)`
  - `tokio::time::sleep(...)` vs `std::thread::sleep(...)`
  - ⚠ **Warning:** make sure you import the right items (`tokio::time`, `tokio::sync`, etc.)
- **Async is “colored”**
  - functions must be declared `async fn` to use `.await`
  - `.await` can only be used inside an `async` function
- **Futures are lazy**
  - they do nothing until they are **awaited** (or polled by the executor)
  - a spawned task is a **Future** that the runtime will poll

PART 2 – ASYNC (CONT.)

- **Task ≠ thread**
  - a Tokio **task** is a lightweight unit of work managed by the runtime
  - whether it runs on 1 OS thread or many depends on the **executor configuration** (current-thread vs multi-thread runtime)
  - tasks can be moved between worker threads by the runtime
- **Cooperative concurrency**
  - tasks must **yield** regularly (typically at `.await` points)
  - the runtime makes progress by polling tasks that are ready
- **⚠ Big warning: never block the runtime**
  - blocking a task can freeze **all** tasks on that worker thread (including the CLI loop!)
  - avoid blocking calls inside async code: `std::thread::sleep`, heavy CPU loops, blocking I/O, etc.
  - if you must run blocking or CPU-bound work:
    - use `tokio::task::spawn_blocking`
    - it runs on a **dedicated blocking thread pool** and does not block the async runtime

PART 2 - TOKIO MPSC

- **mpsc (multi-producer, single-consumer)**
  - tasks communicate by **sending messages**,
  - clear data flow: producer → consumer,
  - no shared mutable state.
- In our case:
  - the **refresh task** sends updates,
  - the **CLI loop** receives and processes them.
- **Alternative: shared memory**
  - `Arc<Mutex<T>>` or `Arc<RwLock<T>>`
  - useful when multiple tasks need direct access to the same state
  - requires careful locking and can introduce contention
- For this workshop, we prefer **channels**:
  - simpler mental model,
  - explicit data flow,
  - well-suited for a CLI event loop.

PART 3 - API WRAPPING AND DEPENDENCY INVERSION

PART 3 – API WRAPPING

- **Goal**: wrap the SNCF API behind a clean and minimal interface.
- We expose only what the application really needs:
  - `fetch_places()` — retrieve and filter places (stop areas),
  - `fetch_journeys()` — retrieve journeys between two places.
- The rest of the application:
  - does not know about HTTP,
  - does not know about JSON payloads,
  - only works with domain types.
- This gives us a **clear boundary** between the external API and our core logic.

PART 3 - DEPENDENCY INVERSION

- Directly calling the HTTP client makes code:
  - hard to test,
  - tightly coupled to the network,
  - dependent on external services.
- Instead, we invert the dependency:
  - define an **HTTPClient trait**,
  - provide a real implementation (`ReqwestClient`),
  - provide a fake one (`FakeClient`) for tests.
- Benefits:
  - test logic in **isolation**,
  - no network access during tests,
  - deterministic and fast test suite.
- **Key principle**:
  - The API layer depends on **abstractions**, not on concrete implementations.
- **Example**:
  - if the SNCF API is no longer HTTP-based (for example **gRPC**),
    (note: the trait name `HTTPClient` would no longer be well suited in that case)
  - we only need a **new client implementation**,
  - the rest of the application remains unchanged.

PART 3 – DATA SERIALIZATION WITH SERDE

- **Serde** is the standard framework in Rust for **serialization and deserialization**.
  - JSON, YAML, TOML, CSV, and more,
  - fast, type-safe, and widely adopted.
- Two common deserialization modes:
  - **Typed**: deserialize into user-defined structs (`#[derive(Deserialize)]`)
  - **Free-form**: deserialize into generic Serde types (e.g. `serde_json::Value`)
- **In this workshop**:
  - we mostly use the **typed approach**,
  - to keep domain logic explicit, safe, and easy to test.

PART 3 – SERDE: SMALL EXAMPLE

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

#[derive(Serialize, Deserialize)]
struct Place {
    id: String,
    name: String,
}

// Deserialize JSON into a Rust type
let place: Place = serde_json::from_str(json_str)?;

// Serialize a Rust type back to JSON
let json = serde_json::to_string(&place)?;
```

PART 3 - TIME HANDLING WITH JIFF

- Dealing with time is hard:
  - time zones,
  - durations,
  - formatting for display.
- We use **jiff** to build a solid time model:
  - parse zoned datetimes from the API,
  - compute journey durations safely,
  - store time-related data in explicit structures.
- Alternatives in the Rust ecosystem:
  - **chrono** — widely used, feature-rich, mature
  - **time** — lower-level, explicit, std-like design
- Choice here:
  - explicit time zones and strong types,
  - clear semantics for dates and durations,
  - a solid foundation for domain logic and UI.

PART 3 – HTTP CLIENT WITH REQWEST

- **reqwest** is the **go-to HTTP client** in the Rust ecosystem
  - widely used and battle-tested,
  - built on top of `hyper`,
  - well integrated with async Rust.
- Supports both **async** and **blocking** APIs:
  - `reqwest::Client` — async (used in this workshop),
  - `reqwest::blocking::Client` — sync, for simple tools.
- In our case:
  - used inside the `ReqwestClient` implementation,
  - configured with a custom **User-Agent**,
  - handles authentication and JSON deserialization.
- Why it fits well here:
  - non-blocking HTTP calls,
  - works naturally with Tokio,
  - easy to replace thanks to dependency inversion.

PART 3 – WHAT CHANGED (AND HOW TO TEST)

- **What we need to implement**:
  - `fetch_journeys()` — call the API, parse zoned datetimes, compute durations, and map results to the domain model
  - `ReqwestClient` — the real HTTP implementation of the client trait
- Most of the changes are located in the **`sncf` crate**.
  - API wrapping,
  - dependency inversion,
  - time handling and domain mapping.
- Some tests are marked as **ignored**:
  - they hit the **live SNCF API**,
  - they require network access and a valid API key.
- To run them:
  - remove the `#[ignore]` attribute
- **Note**:
  - the `todo!()` macro can be used to **temporarily relax the compiler**
  - it allows the code to compile while a part is still missing, until it is properly implemented.

PART 4 - CONFIG AND CLI

PART 4 – CONFIG AND CLI

- **Goal**: configure the application with **start** and **destination** stations.
- Configuration is persisted in **`config.toml`**.
- Startup logic:
  - **No config file**:
    - launch the CLI station picker,
    - we cannot start the refresh task yet (stations are unknown),
    - save the selected stations to `config.toml`.
  - **Config file exists**:
    - for now we still show the same screen (simpler flow),
    - but we also start the background refresh task,
    - it should periodically emit messages in the **logs**.
- This is the first step toward a full CLI dashboard:
  - configuration screen → then live data screen.

PART 4 – CRATES: RATACLI & CROSSTERM

- **RataCLI**
  - a Rust library to build rich **terminal user interfaces** (CLIs)
  - widgets: `Block`, `Paragraph`, `List`, `Table`, etc.
  - layout system to split the screen and render components
  - render loop friendly: draw the UI repeatedly from the current state
- **Crossterm**
  - cross-platform terminal control and input
  - keyboard events, mouse events (optional), terminal resize
  - enables CLI mode:
    - **raw mode** (no line buffering)
    - **alternate screen** (clean full-screen UI)
- How they work together:
  - Crossterm handles terminal + events
  - RataCLI renders widgets into a terminal backend
  - we must always **restore** the terminal on exit (even on panic)

PART 4 – RAW MODE & ALTERNATE SCREEN

- **Raw mode**
  - disables the terminal's default line buffering
  - key presses are sent immediately to the application (no need to press Enter)
  - special keys are exposed as events (`Up`, `Down`, `Esc`, etc.)
- What changes compared to normal mode:
  - no automatic echo of typed characters
  - no terminal-side editing (backspace, arrows, etc.)
  - the application is fully responsible for input handling
- **Alternate screen**
  - switches to a separate, temporary screen buffer
  - the original terminal content is preserved
  - on exit, the previous screen is restored automatically
- Why both are important for CLIs:
  - raw mode enables responsive, interactive input
  - alternate screen provides a clean full-screen UI
  - together, they allow building apps that behave like native tools
- **Note**:
  - stdout / stderr output (`println!`, `eprintln!`) will not work properly anymore,
  - use logging to a file instead (e.g. `tracing`).

PART 4 – MODULAR CLI STRUCTURE (MVP)

- The CLI code is split into small modules to stay maintainable:
  - `app.rs` — application state and update logic
  - `ui.rs` — rendering (widgets + layout)
  - `event.rs` — input/events handling
- This matches an **MVP-like** architecture:
  - **Model**: app state (`app.rs`)
  - **View**: terminal UI (`ui.rs`)
  - **Presenter**: event loop + updates (`event.rs`)
- Benefits:
  - clear separation of concerns,
  - easier to test and reason about,
  - UI can evolve without rewriting everything.

PART 4 – SNAPSHOT TESTING WITH INSTA

- **insta** is a snapshot testing library for Rust.
  - captures the output of a function,
  - stores it as a snapshot,
  - compares future runs against it.
- Why snapshot testing fits CLIs well:
  - CLIs are hard to test with traditional assertions,
  - the rendered output is mostly text-based,
  - snapshots make visual regressions easy to spot.
- In this project:
  - we snapshot the input screen rendering,
  - without starting a real terminal,
  - by testing the UI logic in isolation.
- Workflow:
  - run tests → snapshot is created or compared,
  - review changes explicitly when the UI evolves,
  - accept updates when they are intentional.

PART 4 – WHAT TO DO IN THIS STEP

- Run the tests with `cargo t` and fix failures.
- But this part includes **a lot of code changes**:
  - the most important goal is to **understand the structure** and **play with the code**
  - don’t hesitate to explore:
    - change a widget,
    - tweak navigation,
    - inspect logs and state updates.
- Please ask questions!
  - if something is unclear, stop and ask — we will explain it together
  - the goal is to learn the patterns, not to “speedrun” the commit

PART 5 - CLI AND DASHBOARD

PART 5 – GOAL

- **Goal**: display a real-time dashboard of journeys between the configured **start** and **destination** stations.
- The screen is split into two panels:
    - **Left panel**: journeys list (table)
        - departure date/time,
        - travel duration,
        - number of transfers / changes.
    - **Right panel**: countdown timer
        - time remaining from **now** until the selected departure,
        - big, readable `HH:MM:SS` display.
- **Background refresh task**:
    - periodically fetches journeys from the SNCF API (**every 30 seconds**),
    - sends updated data to the dashboard,
    - the UI refreshes automatically based on received data.

PART 5 – CLI-BIG-TEXT (LARGE ASCII RENDERING)

- **CLI-big-text** is a small helper crate built on top of **RataCLI**.
- You can think of it as a **plugin-like widget**:
  - it renders text using large ASCII-style characters,
  - designed to be easily embedded in a RataCLI layout.
- In our dashboard:
  - used to display the countdown timer,
  - large `HH:MM:SS` format, readable at a glance.

PART 5 – DESKTOP NOTIFICATIONS WITH NOTIFY-RUST

- **notify-rust** is a Rust crate to send **desktop notifications**.
- It integrates with the operating system's notification system (when supported).
- In our dashboard:
    - when the countdown timer reaches **zero**,
    - a notification is sent to the OS,
    - to alert the user that the departure time has arrived.
- The notification is sent **only once**:
    - tracked via internal timer state,
    - to avoid spamming the user.

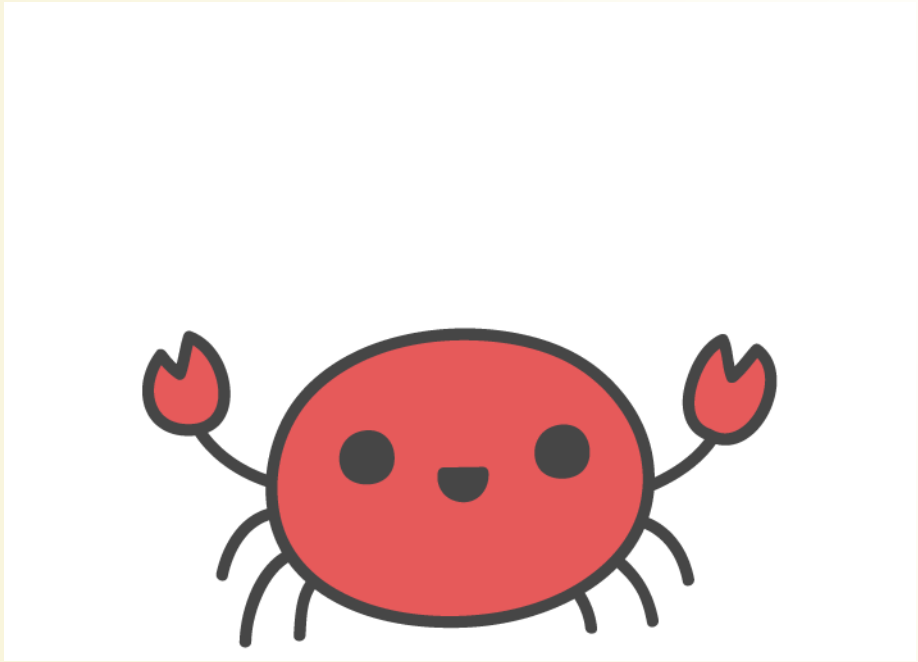
PART 5 – ONE OF THE CHALLENGES

- One important challenge in this part is how to **manage the HTTP client**.
- In **Timer mode** (the dashboard):
  - the HTTP client is currently used **only by the background refresh task**,
  - the CLI loop only consumes the refreshed data (it does not call the API directly).
- This makes one solution tempting:
  - keep the HTTP client entirely inside the task.
- But we want the design to stay flexible:
  - future features may require HTTP calls from other places (CLI actions, details view, manual refresh, etc.),
  - one possible direction is to make the HTTP client a **property of the application state** (`App`),
  - so it can be shared when needed, without duplication.

PART 5 – WHAT TO DO IN THIS STEP

- Run the tests with `cargo t` and fix failures.
- This part introduces **new behaviors and state**:
  - background data refresh,
  - timer and countdown logic,
  - UI updates driven by incoming data.
- One part is intentionally a bit tricky:
  - the HTTP client must be used across asynchronous boundaries,
  - and the code needs to compile and remain correct.
- A suggested approach:
  - first, **make the code compile**, even if it means using multiple client instances,
  - then, step back and think about **a better way to share the client**.
- Please ask questions!
  - this is a good opportunity to discuss ownership, sharing, and async design choices,
  - the goal is to understand the trade-offs, not just the final solution.

THANKS



- Cyril Marin <marin.cyril@gmail.com>
- René Ribaud <rene.ribaud@gmail.com>