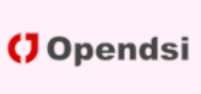
ASYNC RUST AND CLI

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

QUICK POLL

- 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++)
    - 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.

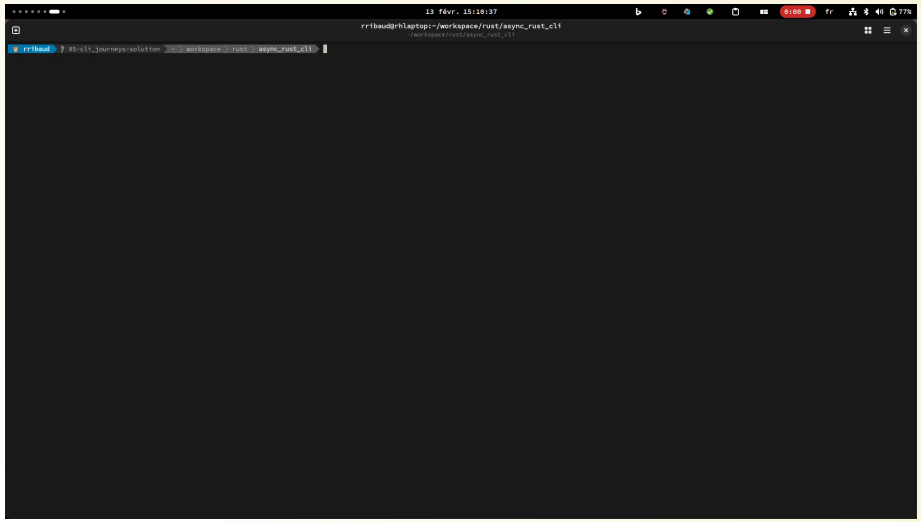
SOME OF THE MAIN DIFFICULTIES (OPTIONAL)

- **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 **~1300 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 – PROJECT LAYOUT AND CORE FILES

- **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 / CLI.
- **Key project files**
  - `main.rs` — binary entry point
    - CLI / application startup,
    - wires configuration, runtime, and top-level logic.
  - `lib.rs` — library root
    - exposes the public API of a crate,
    - contains reusable and testable logic.
  - `Cargo.toml` — project manifest
    - dependencies, features, binaries, metadata.
  - `Cargo.lock` — dependency lockfile
    - exact versions used to ensure reproducible builds.

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
- **Logging with `RUST_LOG`**
  - `RUST_LOG` is an environment variable used to **control log verbosity**.
  - It works with `tracing` (via log compatibility) and `env_logger`-style filters.
  - Examples:
    - `RUST_LOG=info` — show info, warn, and error logs
    - `RUST_LOG=debug` — include debug logs
    - `RUST_LOG=sncf=debug` — debug only the `sncf` crate

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 CLI 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 - CLI FOR PLACES

PART 4 – CLI FOR PLACES

- We start by defining the command structure:
  - a root command (the binary),
  - two subcommands: `places` and `journeys`.
- In this part, we implement only the first subcommand: **`places`**
  - query SNCF places by name (search string),
  - print matching results: **place ID + place name**,
  - useful to discover the IDs needed for the `journeys` command.

PART 4 – CLI WITH CLAP

- **clap** is the de facto standard crate for building command-line interfaces in Rust.
- Key features:
  - argument parsing and validation,
  - subcommands support,
  - auto-generated `--help` and usage messages,
  - type-safe arguments (parsed directly into Rust types).
- We use the **derive macros**:
  - arguments defined as struct fields with attributes.
- Benefits:
  - declarative CLI definition,
  - less boilerplate than manual parsing,
  - clear and maintainable command structure.

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:
- 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 FOR JOURNEYS

PART 5 – CLI FOR JOURNEYS

- **Goal**: implement the **`journeys`** subcommand.
- What it does:
  - call `fetch_journeys()` for the configured route,
  - sort journeys by **departure time**,
  - display a readable list in the terminal.
- Periodic refresh:
  - spawn a background task,
  - fetch new journeys **every 30 seconds**,
  - send results to the main loop via a channel.
- Time handling:
  - use **jiff** to compute timestamps,
  - compute remaining time until departure for each journey.

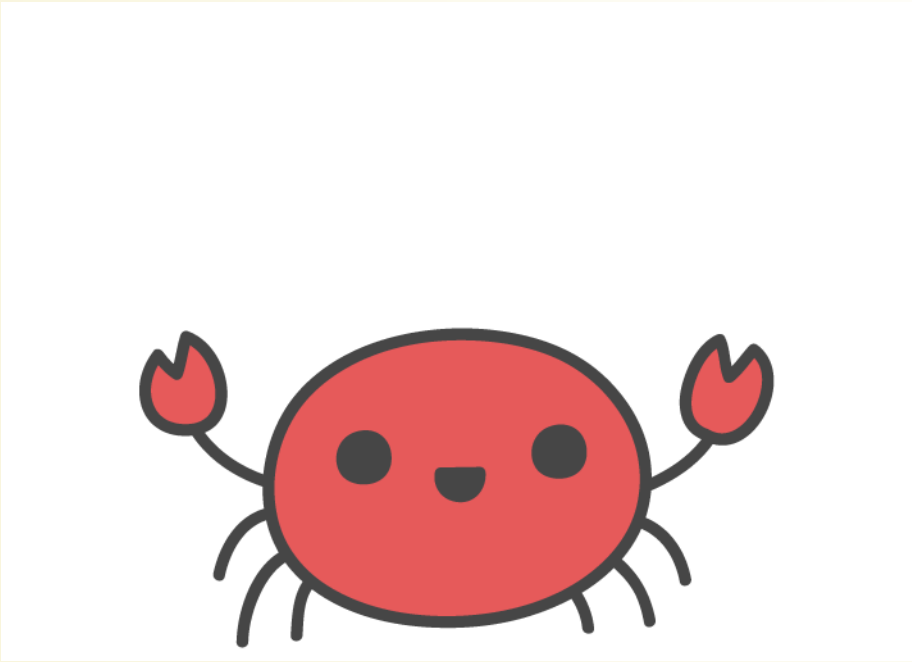
PART 5 – WHAT TO DO IN THIS STEP

- 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:
- Error handling is part of the exercise:
  - handle the case where the user provides an **invalid place ID**,
  - return a clear and user-friendly error instead of crashing.
- 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

OTHER REPOSITORIES

- https://github.com/uggla/async_rust_tui
  - Same project and learning approach for the first parts.
  - Parts **4 and 5** diverge: this repository focuses on building a **full TUI**.
  - Explores async Rust, API wrapping, dependency inversion, and terminal UI patterns.
- https://github.com/uggla/bevy_university
  - Uses the **same pedagogical approach** (progressive steps, tests, exploration),
  - but applies it to a **different domain**: game development with **Bevy**.
  - Focuses on ECS concepts, rendering, and real-time systems in Rust.

THANKS



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