Un Grand MERCI à nos sponsors 2026

CGI

open

KAIZEN SOLUTIONS

aws

CRITEO

Moody's

clever cloud

AVISTO

KLS GROUP

kelkoogroup

VISEO
POSITIVE DIGITAL MAKERS

alma

sopra steria

PeriScop

enalean
software engineering is now

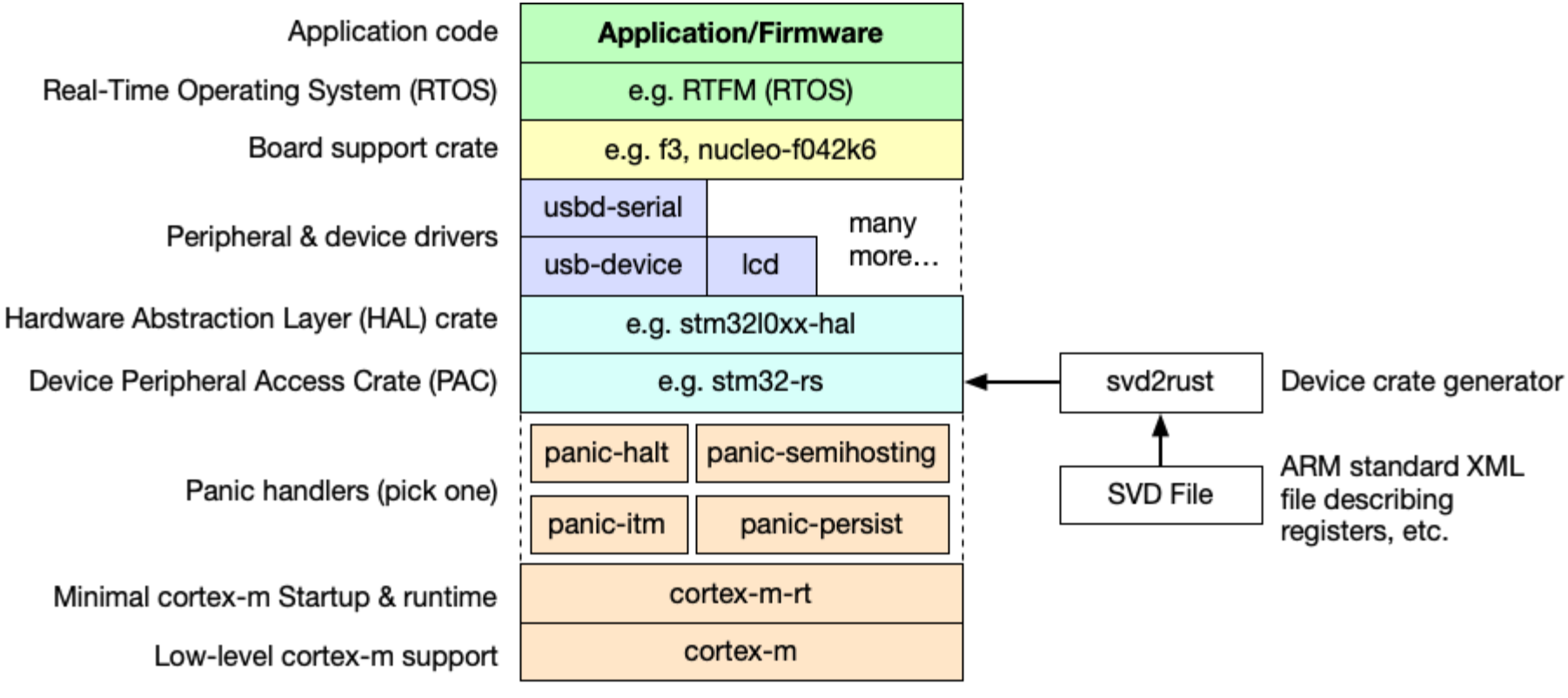
HACK YOUR JOB

PingID

zenika

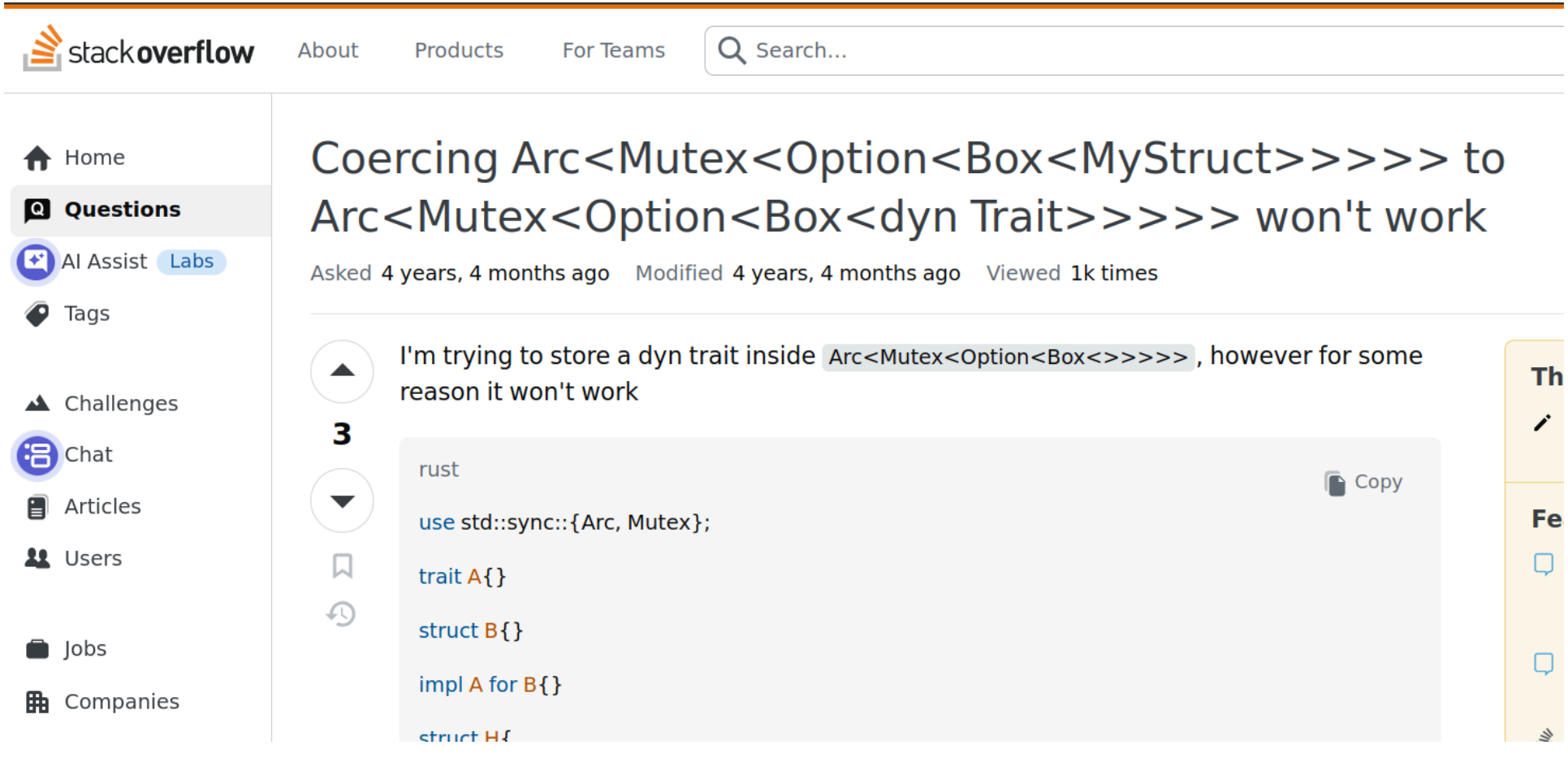
Bienvenue ! 👋

Rust 🦀 est connu pour être un langage qui favoriserait le "bas-niveau"...



Pour des masochistes férus d'optimisations mémoire...

À la syntaxe souvent qualifiée d'exigeante, quand ce n'est pas carrément "cryptique"... 🤯



Et si je vous disais que tout ceci est probablement vrai, mais que ce n'est pas pour ça que Rust m'intéresse ?

🦀Rust, pour des applications métiers 🦀

Un système typal expressif et multi-paradigmes

La base: Structs, et "Newtypes"

Un cas ultra-classique:

```
pub struct User {
    pub email: String,
}

fn ma_fonction(user: User) {
  if !user.email.contains("@") {
     // Gestion de l'erreur, etc.
  }
  // On continue...

  // Possible, mais pas voulu -> Bug !
  let city: String = user.email
}
```

Encoder la logique métier, le rêve ! 😌

```rust
use crate::domain::UserEmail;

pub struct User {
    // Attendez…c'est quoi ça ?
    pub email: UserEmail,
}
```

On "impl"émente la logique pour traiter les cas :

```rust
use validator::ValidateEmail;

#[derive(Debug)]
pub struct UserEmail(String);

impl UserEmail {
    pub fn parse(s: String) -> Result<UserEmail, String> {
        if s.validate_email() {
            Ok(Self(s))
        } else {
            Err(format!("{} is not a valid user email", s))
        }
    }
}
```

Comment on valide ça ? Avec des tests ! 🤩

```rust
#[cfg(test)]
mod tests {
    #[test]
    fn email_missing_at_symbol_is_rejected() {
        let email = "stephanedomain.com".to_string();
        assert_err!(UserEmail::parse(email));
    }

    #[test]
    fn email_missing_subject_is_rejected() {
        let email = "@domain.com".to_string();
        assert_err!(UserEmail::parse(email));
    }
}
```

💥 Un newtype n'est pas égal à un autre type !

```
pub struct Address(String);
pub struct UserEmail(String);

pub struct User { pub email: UserEmail }

let my_user = User {
  // Erreur ! Pas un UserEmail, mais un String !
  email: "steph@mydomain.com"
}

let my_user = User {
  // Erreur ! Pas un UserEmail, mais un Address !
  email: Address("steph@mydomain.com")
}
```

Il faut utiliser le newtype prévu :

```rust
use crate::domain::UserEmail;

pub struct User {
    pub email: UserEmail,
}

let my_user = User {
  // Ok 👍
  email: UserEmail::parse("steph@mydomain.com")?;
}
// Note: Pour l'instant le `?` c'est "magique" 🪄
```

Parse, don't validate

2019-11-05 • functional programming, haskell, types

Historically, I've struggled to find a concise, simple way to explain what it means to practice type-driven design. Too often, when someone asks me "How did you come up with this approach?" I find I can't give them a satisfying answer. I know it didn't just come to me in a vision—I have an iterative design process that doesn't require plucking the "right" approach out of thin air—yet I haven't been very successful in communicating that process to others.

However, about a month ago, I was reflecting on Twitter about the differences I experienced parsing JSON in statically- and dynamically-typed languages, and finally, I realized what I was looking for. Now I have a single, snappy slogan that encapsulates what type-driven design means to me, and better yet, it's only three words long:

Parse, don't validate.

Impureim sandwich by Mark Seemann

Pronounced 'impurium sandwich'.

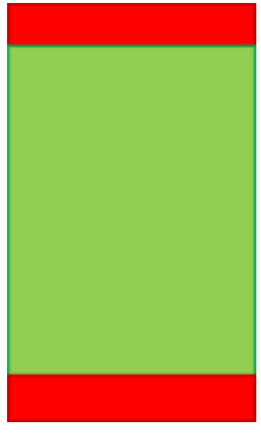
Since January 2017 I've been singing the praise of the *impure/pure/impure* sandwich, but I've never published an article that defines the term. I intend this article to remedy the situation.

Functional architecture

In a functional architecture, pure functions can't call impure actions. On the other hand, as Simon Peyton Jones observed in a lecture, *observing the result of pure computation is a side-effect*. In practical terms, *executing* a pure function is also impure, because it happens non-deterministically. Thus, even for a piece of software written in a functional style, the entry point must be impure.

While pure functions can't call impure actions, there's no rule to prevent the obverse. Impure actions *can* call pure functions.

Therefore, the best we can ever hope to achieve is an impure entry point that calls pure code and impurely reports the result from the pure function.



The flow of code here goes from top to bottom:

1. Gather data from impure sources.
2. Call a pure function with that data.
3. Change state (including user interface) based on return value from pure function.

This is the *impure/pure/impure* sandwich.

L'A et l'Ω du Type Driven Development:

Le Pattern Matching

Exemple avec un enum (tout bête) :

```
pub enum LoginError {
    AuthenticationError(String),
    SSOError(String),
    UnexpectedError(String),
}
```

Si on veut pouvoir l'utiliser, on peut s'aider de `match`

```rust
pub enum LoginError {
    AuthenticationError(String),
    SSOError(String),
    UnexpectedError(String),
}


fn my_fn(e: LoginError) {
match e {}
}
```

missing match arm: `AuthenticationError(_)`, `SSOError(_)` and `UnexpectedError(_)` not covered (rust-analyzer E0004)

https://doc.rust-lang.org/stable/error_codes/E0004.html

non-exhaustive patterns: `LoginError::AuthenticationError(_)`, `LoginError::SSOError(_)` and `LoginError::UnexpectedError(_)` not covered
the matched value is of type `LoginError`

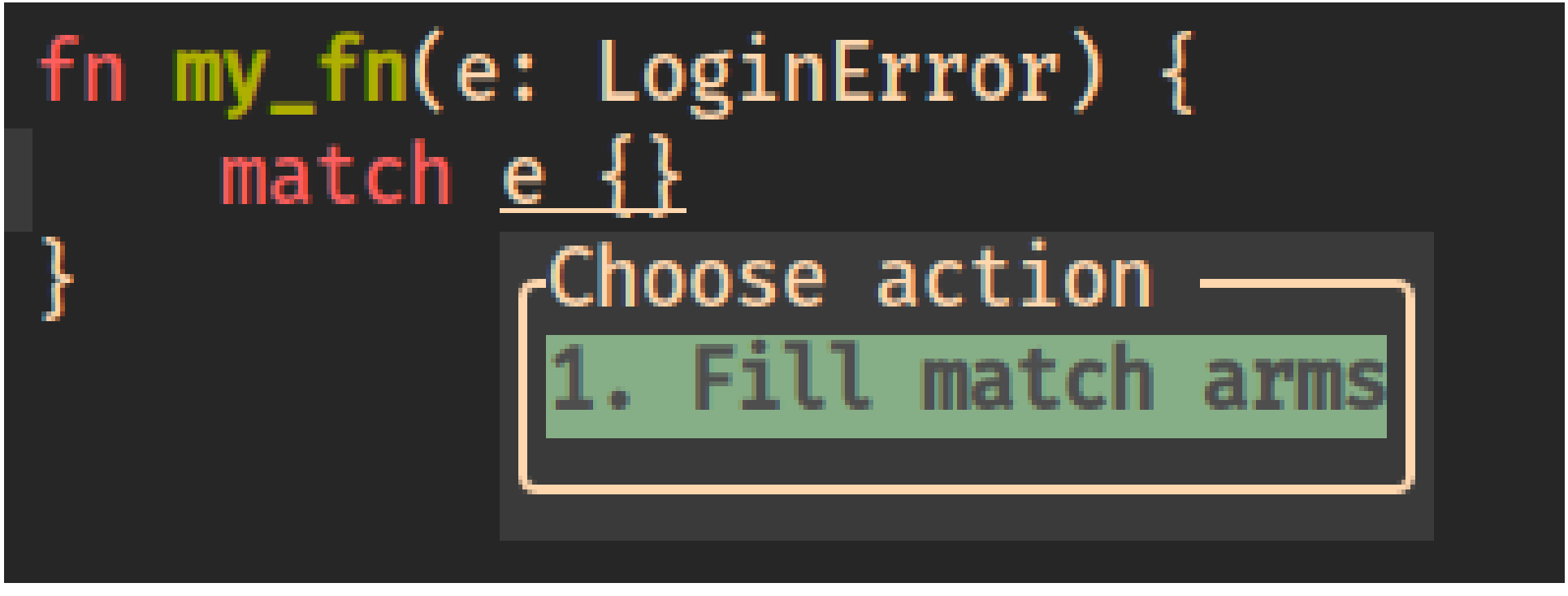
Related information:

  * main.rs#401,10: `LoginError` defined here
  * main.rs#409,8: ensure that all possible cases are being handled by adding a match arm with a wildcard pattern, a match arm with multiple or-patterns as shown, or multiple match arms: ` {
    LoginError::AuthenticationError(_) | LoginError::SSOError(_) | LoginError::UnexpectedError(_) ⇒ todo!(),
}`

 (rustc E0004)

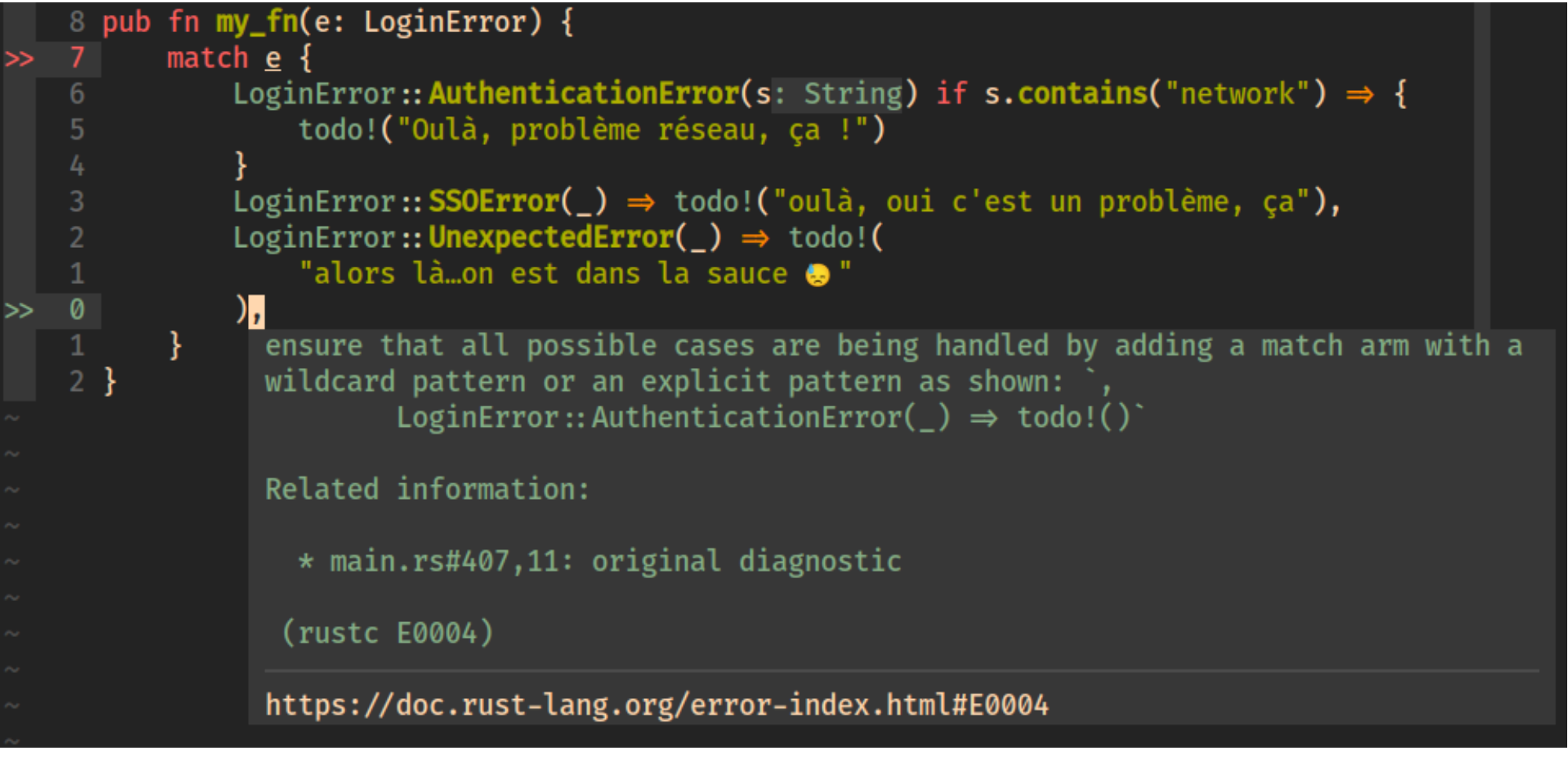
https://doc.rust-lang.org/error-index.html#E0004

Fort heureusement on a ce qu'il faut avec le LSP de Rust 😌



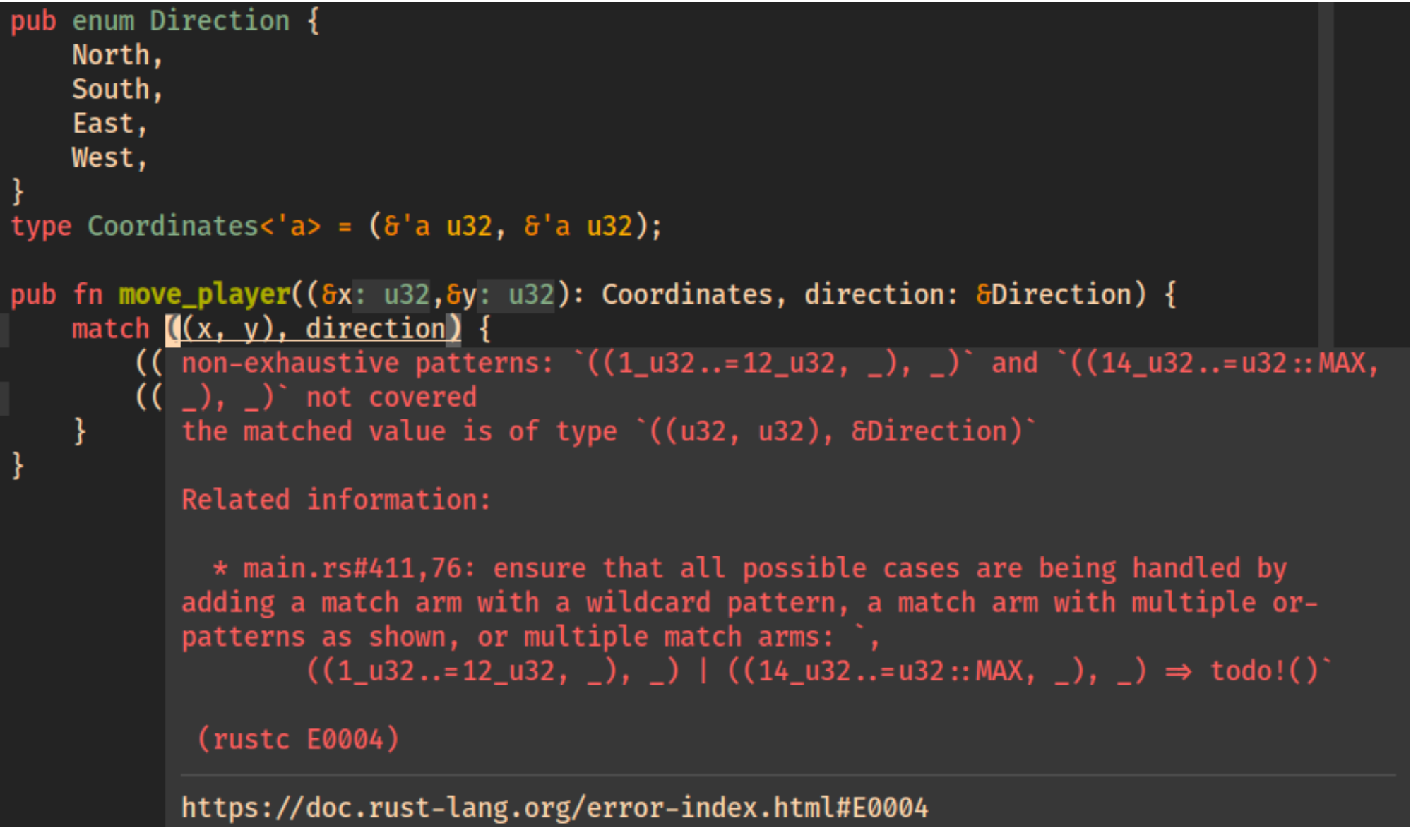
Et y a plus qu'à remplir !

```rust
pub fn my_fn(e: LoginError) {
    match e {
        LoginError::AuthenticationError(_) ⇒ todo!("y a plus qu'à !"),
        LoginError::SSOError(_) ⇒ todo!("oulà, oui c'est un problème, ça"),
        LoginError::UnexpectedError(_) ⇒ todo!("alors là…on est dans la sauce 😓"),
    }
}
```


8 pub fn my_fn(e: LoginError) {
7 match e {
6 LoginError::AuthenticationError(s: String) if s.contains("network") ⇒ {
5 todo!("Oulà, problème réseau, ça !")
4 }
3 LoginError::SSOError(_) ⇒ todo!("oulà, oui c'est un problème, ça"),
2 LoginError::UnexpectedError(_) ⇒ todo!(
1 "alors là…on est dans la sauce 😓"
0),
1 }
2 }
ensure that all possible cases are being handled by adding a match arm with a
wildcard pattern or an explicit pattern as shown: `,
LoginError::AuthenticationError(_) ⇒ todo!()`
Related information:
* main.rs#407,11: original diagnostic
(rustc E0004)
https://doc.rust-lang.org/error-index.html#E0004

Et on peut aller très très loin !

```rust
pub enum Direction {
    North,
    South,
    East,
    West,
}
type Coordinates<'a> = (&'a u32, &'a u32);

pub fn move_player((&x: u32,&y: u32): Coordinates, direction: &Direction) {
    match ((x, y), direction) {
        ((0,0), _) ⇒  todo!("Origine du plan, rien à faire"),
        ((13,10), Direction::South) ⇒ todo!("Ah non y a l'océan par-là !"),
    }
}
```


pub enum Direction {
North,
South,
East,
West,
}
type Coordinates<'a> = (&'a u32, &'a u32);
pub fn move_player((&x: u32,&y: u32): Coordinates, direction: &Direction) {
match ((x, y), direction) {
((non-exhaustive patterns: `((1_u32..=12_u32, _), _)` and `((14_u32..=u32::MAX,
((_), _)` not covered
} the matched value is of type `((u32, u32), &Direction)`
}
Related information:
* main.rs#411,76: ensure that all possible cases are being handled by
adding a match arm with a wildcard pattern, a match arm with multiple or-
patterns as shown, or multiple match arms: `,
((1_u32..=12_u32, _), _) | ((14_u32..=u32::MAX, _), _) ⇒ todo!()`
(rustc E0004)
https://doc.rust-lang.org/error-index.html#E0004

Finalement, tout s'arrange 😉

```rust
pub enum Direction {
    North,
    South,
    East,
    West,
}
type Coordinates<'a> = (&'a u32, &'a u32);

pub fn move_player((&x: u32,&y: u32): Coordinates, direction: &Direction) {
    match ((x, y), direction) {
        ((0,0), _) ⇒  todo!("Origine du plan, rien à faire"),
        ((13,10), Direction::South) ⇒ todo!("Ah non y a l'océan par-là !"),
        ((_,_), _) ⇒ todo!("Autres cas à traiter ^^")
    }
}
```

Les "Boîtes" de Rust

"Mais, attends, c'est quoi Result, Ok, et Err ?"

```rust
use validator::ValidateEmail;

#[derive(Debug)]
pub struct UserEmail(String);

impl UserEmail {
    pub fn parse(s: String) -> Result<UserEmail, String> {
        if s.validate_email() {
            Ok(Self(s))
        } else {
            Err(format!("{} is not a valid user email", s))
        }
    }
}
```

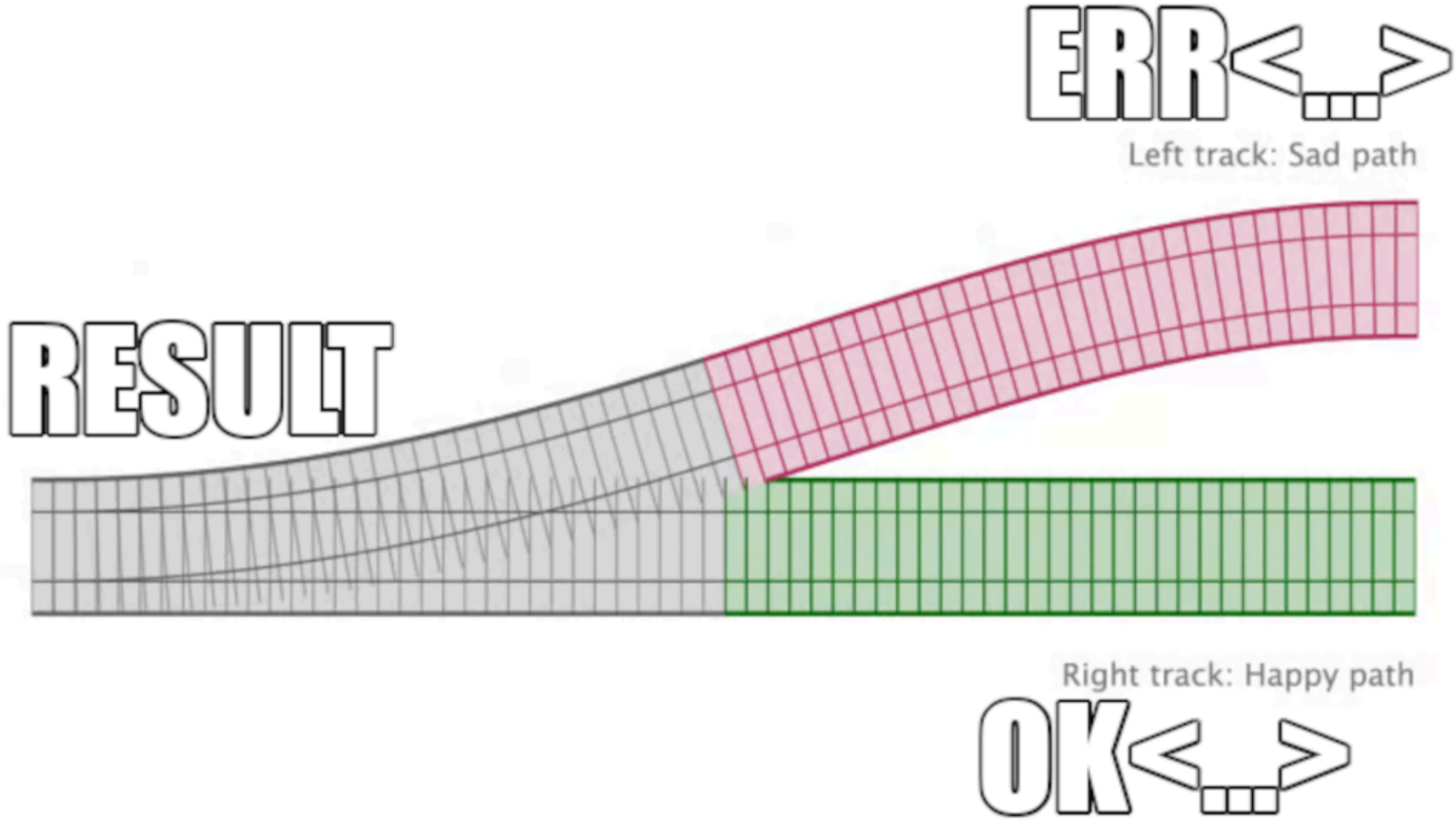

Skip
17
QU'EST CE QUE LA "BILLION DOLLAR MISTAKE" ?
0
Answers
RENOMMER ECMASCRIPT EN JAVASCRIPT
LES MISES À JOUR WINDOWS
L'AJOUT DES HOOKS DANS REACT
L'AJOUT DE NULL À ALGOL W
imgflip.com

Speaking at a software conference in 2009, Tony Hoare hyperbolically apologized for inventing the null reference: [26][27]

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.[28]

L'ajout de `null` à Algol W, et sa réplication dans (presque) tous les languages qui l'ont suivi


NIL !
NIL !
NIL !



source: https://blog.logrocket.com/javascript-either-monad-error-handling/

dans Rust">

Un Result étant un Enum (`Ok` ou `Err`), on peut le `match`

```
/// validate_credentials:
///   (Credentials)-> Result<Uuid, LoginError>
match validate_credentials(credentials).await {
}
```

Et y a plus qu'à remplir !

```
match validate_credentials(credentials).await {
    Ok(_) => todo!("rajouter le cas où tout est ok"),
    Err(_) => todo!("rajouter la gestion d'erreur"),
}
```

Finalement, on met le code final:

```
match validate_credentials(credentials).await {
  Ok(user_id) => {
    return HttpResponse::SeeOther()
      .insert_header((LOCATION, "/admin/dashboard"))
      .finish());
  }
  Err(e) => {
    return login_redirect(LoginError::AuthError(e));
  }
}
```

"On est obligés de gérer à chaque fois `Ok` et `Err` ?"

Non, y a `.unwrap()` pour ça

```
/// In this case, we ALWAYS have an IP Address
let port = listener.local_addr().unwrap().port();
```

Mais pourquoi qu'on s'inflige le `match`, alors ?

Parce que l'alternative est... pire.

Et même les plus gros se font avoir :

When the bad file with more than 200 features was propagated to our servers, this limit was hit — resulting in the system panicking. The FL2 Rust code that makes the check and was the source of the unhandled error is shown below:

```
71      /// Fetch edge features based on `input` struct into [`Features`] buffer.
72      pub fn fetch_features(
73          &mut self,
74          input: &dyn BotsInput,
75          features: &mut Features,
76      ) -> Result<(), (ErrorFlags, i32)> {
77          // update features checksum (lower 32 bits) and copy edge feature names
78          features.checksum &= 0xFFFF_FFFF_0000_0000;
79          features.checksum |= u64::from(self.config.checksum);
80          let (feature_values, _) = features
81              .append_with_names(&self.config.feature_names)
82              .unwrap();
```

This resulted in the following panic which in turn resulted in a 5xx error:

```
thread fl2_worker_thread panicked: called Result::unwrap() on an Err value
```

Réservez `.unwrap()` pour le code de test et apprenez à propager ou gérer les `Result`

```
fn my_function() -> Result<PortNumber, Error> {
  /// In this case, we SHOULD have an IP address.
  /// If not, bubble up the error upwards
  let port = listener.local_addr()?.port();
  return port;
}
```

D'ailleurs, en parlant d'erreurs...

`anyhow` et `thiserror`, les jumeaux surdoués de la gestion d'erreur.

`thiserror`, les erreurs faciles à décrire :

```rust
use thiserror::Error;
#[derive(Error, Debug)]
pub enum DataStoreError {
  #[error("data store disconnected")]
  Disconnect(#[from] io::Error),
  #[error("the data for key `{0}` is not available")]
  Redaction(String),
  #[error("invalid ({expected:?}, found {found:?})")]
  InvalidHeader {
      expected: String,
      found: String,
  },
  #[error("unknown data store error")]
  Unknown }
```

anyhow, les erreurs faciles à créer et propager :

```rust
use anyhow::{Context, Result};

// Le Result ne précise même plus le type d'erreur
// car ce sera un `anyhow::Error` 👌
fn main() -> Result<()> {

  // .context permet d'emballer l'erreur avec…du contexte 😁
  it.detach().context("Failed to detach the thing")?;

  // L'utilisation de `?` veut simplement dire:
  // Si c'est une `Err` alors on bubble up !
  // Sinon on "ouvre" la boîte `Ok` et on prend son contenu
  let content = std::fs::read(path)?;
}
```

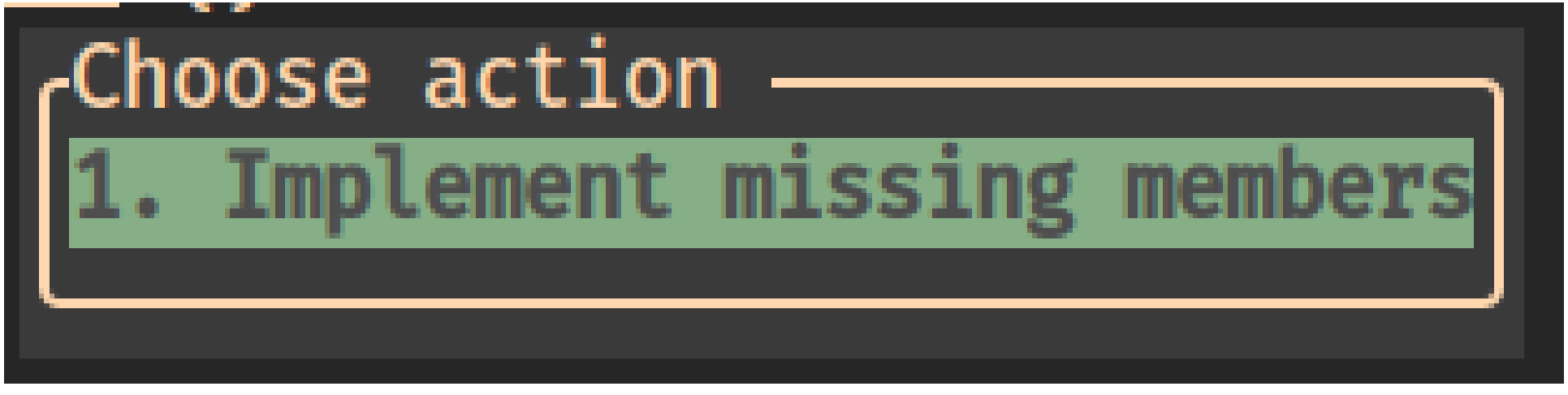
Traits, la POO turbo-chargée

```rust
pub struct RecipientEmail(pub String);
pub struct SenderEmail(pub String);

#[allow(async_fn_in_trait)]
pub trait PaymentProcessor {
    type Error: std::error::Error;

    async fn send_payment(
        recipient_email: &RecipientEmail,
        sender: &SenderEmail,
    ) -> Result<(), Self::Error>;
}

pub struct MyPaymentProcessor {}

impl PaymentProcessor for MyPaymentProcessor {}
```

```
not all trait items implemented, missing: `Error`, `send_payment`
missing `Error`, `send_payment` in implementation

Related information:

  * main.rs#421,5: `Error` from trait
  * main.rs#423,5: `send_payment` from trait
  * main.rs#431,47: implement the missing item: `type Error = /* Type */;`:
`type Error = /* Type */;
`
  * main.rs#431,47: implement the missing item: `async fn send_payment(_:
&RecipientEmail, _: &SenderEmail) -> std::result::Result<(), <Self as
PaymentProcessor>::Error> { todo!() }`: `async fn send_payment(_:
&RecipientEmail, _: &SenderEmail) -> std::result::Result<(), <Self as
PaymentProcessor>::Error> { todo!() }
`


 (rustc E0046)
```

https://doc.rust-lang.org/error-index.html#E0046

Comme pour les enums, on peut utiliser le puissant LSP de Rust



```rust
#[allow(async_fn_in_trait)]
pub trait PaymentProcessor {
    type Error: std::error::Error;

    async fn send_payment(
        recipient_email: &RecipientEmail,
        sender: &SenderEmail,
    ) -> Result<(), Self::Error>;
}

pub struct MyPaymentProcessor {}

#[derive(Debug, Error)]
pub enum PaymentError {
    #[error("Network error while connecting: {0}")]
    NetworkError(String),
    #[error("Authentication rejected")]
    AuthenticationError,
    #[error("Woah, never saw that one before !")]
    UnexpectedError,
}

impl PaymentProcessor for MyPaymentProcessor {
    type Error = PaymentError;

    async fn send_payment(
        recipient_email: &RecipientEmail,
        sender: &SenderEmail,
    ) -> Result<(), Self::Error> {
        todo!("Send payment instructions from {sender} {recipient_email} go here !")
    }
}
```

Les traits "standards"

Un exemple simple: J'ai une `LoginError`, et je veux la déboguer

Une "fonction" super pratique: `dbg!(my_error)`

```
let my_error = LoginError::AuthError("Oulà !");
dbg!(&my_error)
```

Sauf que LoginError n'est pas un type élémentaire...

```
`LoginError` doesn't implement `std::fmt::Debug`
add `#[derive(Debug)]` to `LoginError` or manually `impl
std::fmt::Debug for LoginError`

Related information:

  * error.rs#53,18: required by a bound in `Error`
  * post.rs#18,1: consider annotating `LoginError` with
    `#[derive(Debug)]`

 (rustc E0277)
--------------------------------------------------------------------------------
https://doc.rust-lang.org/error-index.html#E0277
```

Une solution, simple et bien pratique dans 90% des cas : `#[derive()]`

```
#[derive(Debug)]
pub enum LoginError {
    AuthError(Error),
    UnexpectedError(Error),
}
```

Ok pour les cas de base, mais implémenter explicitement `Debug` (ou autre trait) ?

Encore une fois, le LSP est là pour nous aider 😍

```
pub enum LoginError {
    AuthError(String),
    UnexpectedError(String),
}

impl std::fmt::Debug for LoginError {}
                ┌Choose action ──────────────────┐
                │1. Implement missing members    │
                └────────────────────────────────┘
```

On obtient une implémentation "par defaut", qu'on peut customiser 🤤

```
impl std::fmt::Debug for LoginError {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    match self {
      Self::AuthError(arg0) =>
          f.debug_tuple("AuthError").field(arg0).finish(),
      Self::UnexpectedError(arg0) =>
          f.debug_tuple("UnexpectedError").field(arg0).finish(
    }
  }
}
```

Et on y met ce qu'on veut

(tant qu'on respecte le Trait, bien sûr)

```
impl std::fmt::Debug for LoginError {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        writeln!(f, "{}\n", e)?;
        let mut current = e.source();
        while let Some(cause) = current {
            writeln!(f, "Caused by:\n\t{}", cause)?;
            current = cause.source();
        }
        Ok(())
    }
}
```

Ce qui nous permet d'avoir un détail de l'erreur bien plus clair en debug :

```
Failed to log in user.
Caused by:
    Failed to retrieve user informations
Caused by:
    error returned from database
Caused by:
    table "user" does not exist
```

La "blanket implementation" d'un Trait pour un Type

Idée: Ça serait pas mal de facilement "convertir" un type en un autre 🤔

```
struct FormData {
    key: String,
}

let secure_key: SecurityKey = form.key
  .try_into()
  .map_err(error_400)?;
```

`SecurityKey` a une *blanket implementation* pour le type `String` 💡

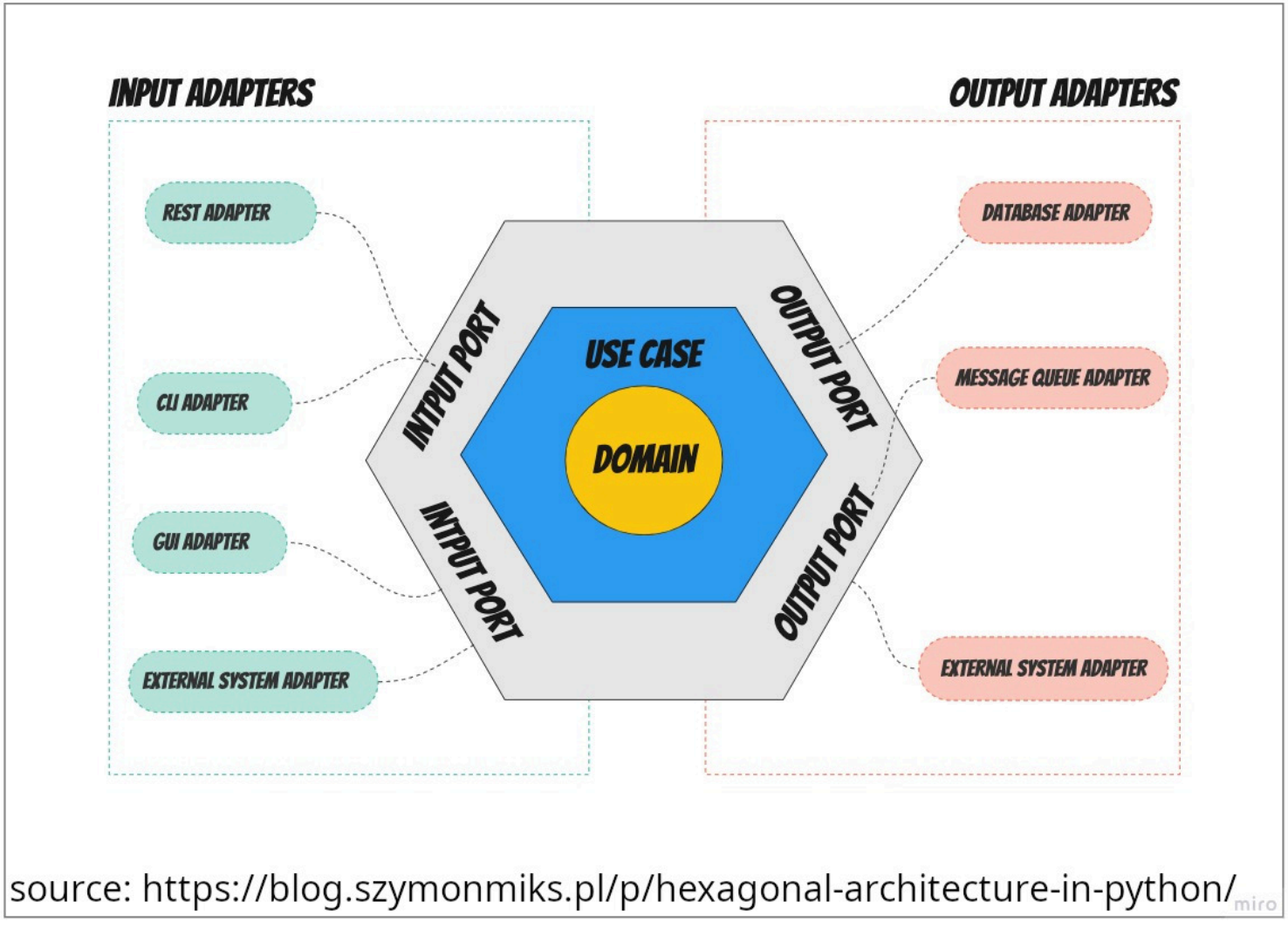
```rust
impl TryFrom<String> for SecurityKey {
type Error = anyhow::Error;
fn try_from(s: String) -> Result<Self, Self::Error> {
    if s.is_empty() {
        anyhow::bail!("The security key cannot be empty.")
    }
    let min_length = 50;
    if s.len() < min_length {
        anyhow::bail!(r#"The security key must be
        longer that {min_length} characters"#)
    }
    Ok(Self(s))
}}
```

Et des types/traits/boîtes comme ça, vous en avez PLEIN, dans Rust 🤩

- `Option` (`Some<…>` ou `None`)
- `Debug` vs `Display`
- `Deref`
- `Drop`

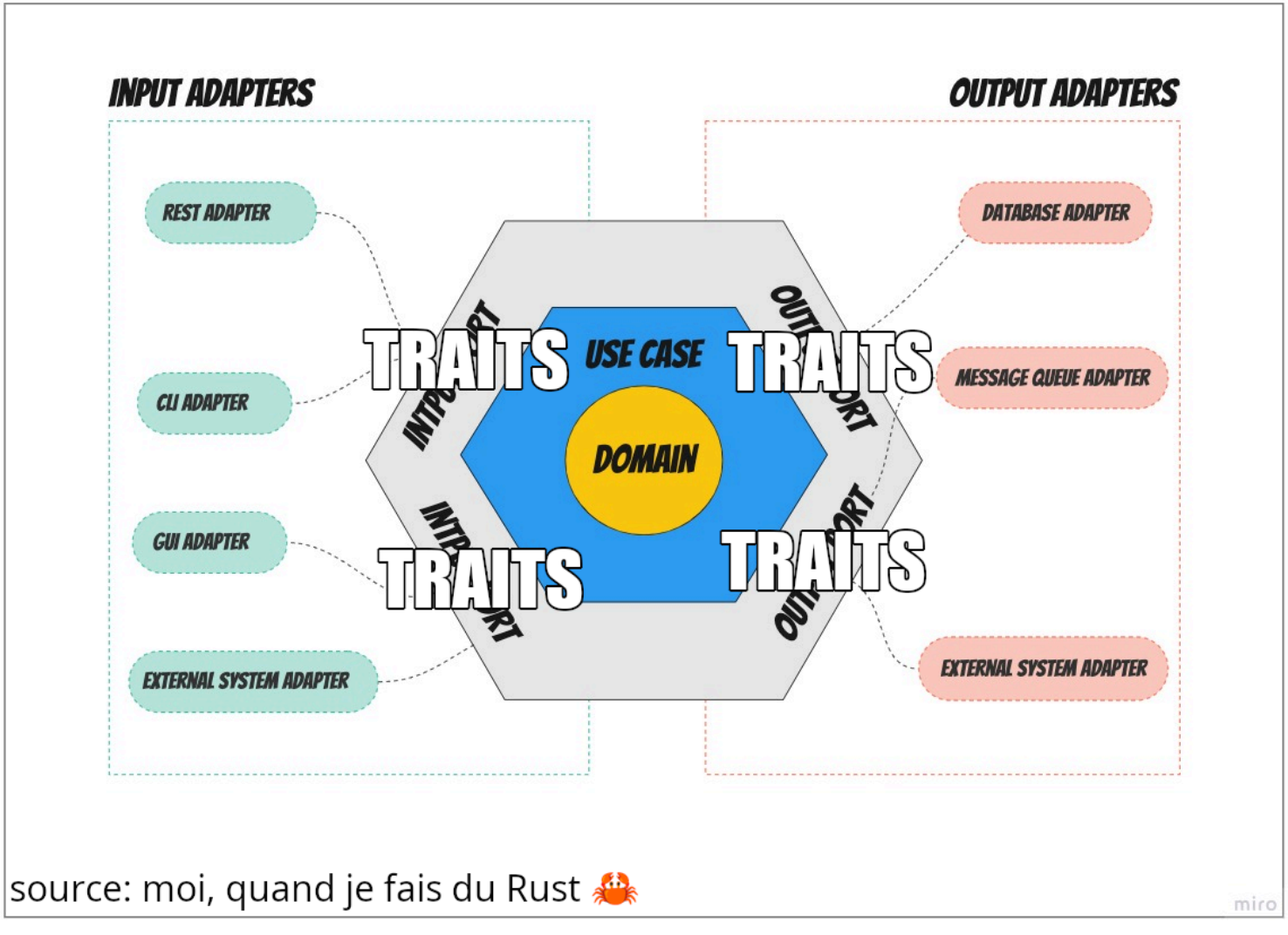
Architecture Hexagonale

Rappel à toutes fins utiles:



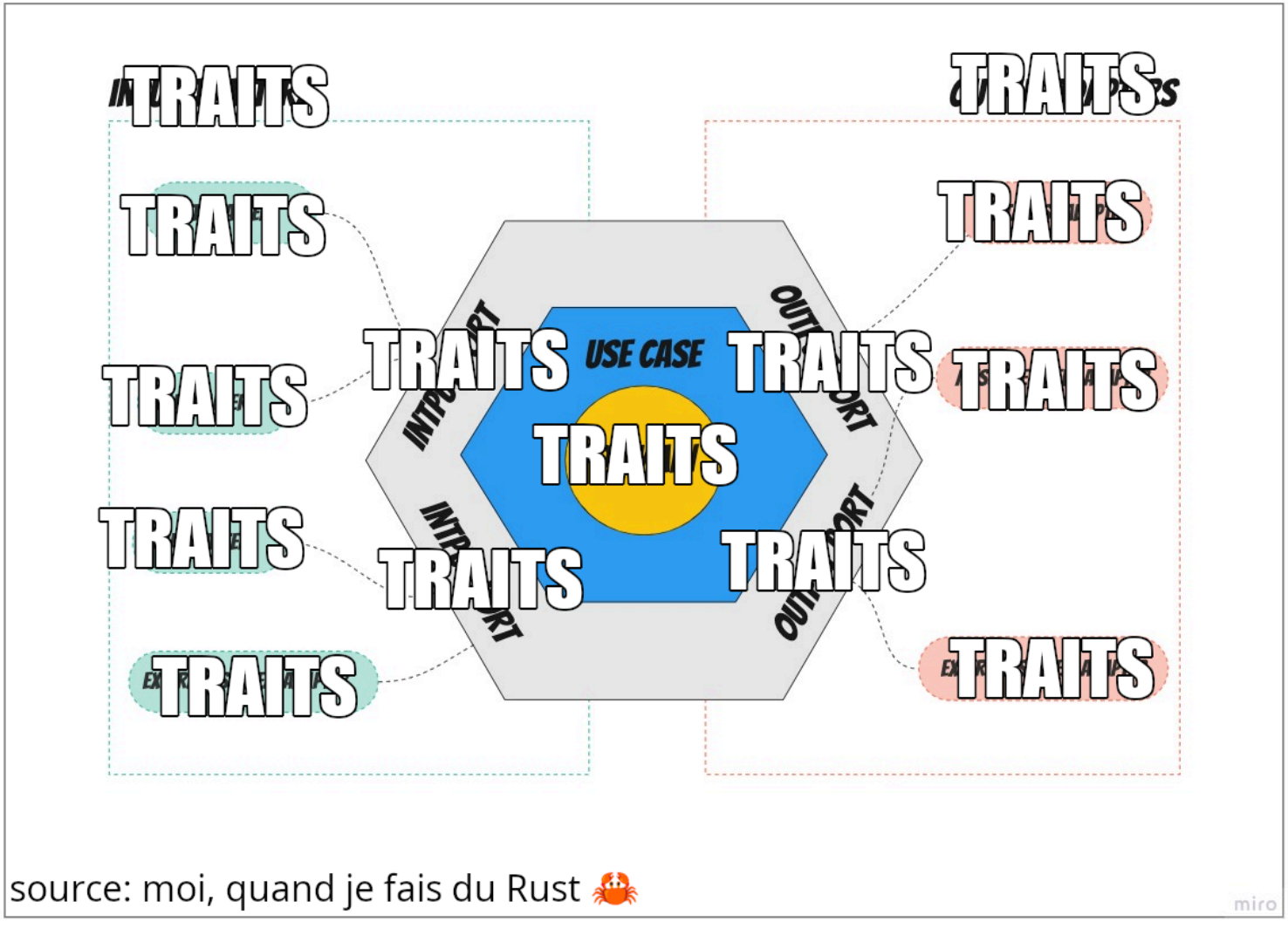
source: https://blog.szymonmiks.pl/p/hexagonal-architecture-in-python/

Donc, si vous avez bien suivi…



source: moi, quand je fais du Rust 🦀

Mais, en vrai…



source: moi, quand je fais du Rust 🦀

Et c'est ok !

```
fn process_data<F, B>(foo: F, bar: B) -> String
where
    F: Clone + Debug,
    B: Display + PartialEq,
{
    // Function implementation
    format!("{:?} - {}", foo, bar)
}
```

⬢ + 🦀 = 💓

```rust
pub async fn create_author<AR: AuthorRepository>(
    State(state): State<AppState<AR>>,
    Json(body): Json<CreateAuthorHttpRequestBody>,
) -> Result<ApiSuccess<CreateAuthorResponseData>, ApiError> {
    let domain_req = body.try_into_domain()?;
    state
        .author_repo
        .create_author(&domain_req)
        .await
        .map_err(ApiError::from)
        .map(|ref author| ApiSuccess::new(StatusCode::CREATED, author.into()))
}
```

source: https://www.howtocodeit.com/guides/master-hexagonal-architecture-in-rust

Oh my.

Isn't it beautiful?

Doesn't your nervous system feel calmer to behold it?

Un écosystème de librairies applicatives solide, complet, et agréablement documenté

Il faut qu'on parle de `sqlx`

```rust
#[tracing::instrument(name = "Retrieve confirmed subscribers from the database", skip(pool))]
pub async fn get_confirmed_subscribers(
    pool: &PgPool,
) -> Result<Vec<Result<ConfirmedSubscriber, anyhow:
    let confirmed_subscribers: Vec<Result<Confirmed
        r#"
        SELECT email from subscriptions WHERE statsu = 'confirmed';
        "#,
    )
    .fetch_all(pool)
    .await
    .unwrap()
    .into_iter()
    .map(|r| match SubscriberEmail::parse(r.email) {
        Ok(e: SubscriberEmail) => Ok(ConfirmedSubscriber { email: e }),
        Err(error: String) => Err(anyhow::anyhow!(error)),
    })
    .collect();

    Ok(confirmed_subscribers)
}
```

error returned from database: column "statsu" does not exist (rust-analyzer macro-error)

https://rust-analyzer.github.io/book/diagnostics.html#macro-error

```rust
#[tracing::instrument(name = "Retrieve presentations from the database", skip(pool))]
pub async fn get_presentations(
    pool: &PgPool,
    product_id: Uuid,
) -> Result<Vec<PresentationSimple>, anyhow::Error> {
    let presentations: Vec<{unknown}> = sqlx::query!(
        r#"
        SELECT Q.id, Q.unit_name, Q.unit_size, Q.packaging_name, Q.packaging_factor,
            Q.strength_unit_name, Q.strength_unit_factor, Q.created_at, Q.updated_at
        FROM presentations Q
        LEFT OUTER JOIN products_presentations_crosstable X
            ON X.presentation_id = Q.id
        LEFT OUTER JOIN products P
            ON P.id = X.product_id
        WHERE P.id = $1;
        "#,
        product_id
    )
    .fetch_all(pool)
    .await
    .unwrap()
    .into_iter()
    .map(|r| PresentationSimple {
        id: r.id,
        unit_name: r.unit_name,
        unit_size: r.unit_size as u16,
        packaging_name: r.packaging_name,
        packaging_factor: r.packaging_factor as u16,
        strength_unit_name: r.strength_unit_name,
        strength_unit_factor: r.strength_unit_factor as u16,
        created_at: r.created_at,
        updated_at: r.updated_at,
    })
    .collect();
```

Les autres poids lourds : serde, tera, chrono, et bien d'autres !

Les Frameworks haut-niveau de Rust 🦀

Les vénérables axum et actix-web

Axum

To build the OpenAPI spec of the whole application, we create a struct in `src/api/v1.rs` and add the `#[derive(OpenApi)]` macro to it:

```
#[derive(OpenApi)]
#[openapi(
    paths(
        handlers::hello::hello,
    ),
    components(
        schemas(

        ),
    ),
    tags(
        (name = "hello", description = "Hello"),
    ),
    servers(
        (url = "/v1", description = "Local server"),
    ),
)]
pub struct ApiDoc;
```

Actix-web

JSON

`Json<T>` allows deserialization of a request body into a struct. To extract typed information from a request's body, the type `T` must implement `serde::Deserialize`.

```
use actix_web::{post, web, App, HttpServer, Result};
use serde::Deserialize;

#[derive(Deserialize)]
struct Info {
    username: String,
}

/// deserialize `Info` from request's body
#[post("/submit")]
async fn submit(info: web::Json<Info>) -> Result<String> {
    Ok(format!("Welcome {}!", info.username))
}
```

Mais le monde des (c)rustacés va vite, très vite…

Dioxus, le framework isomorphique qui monte !

Introduction

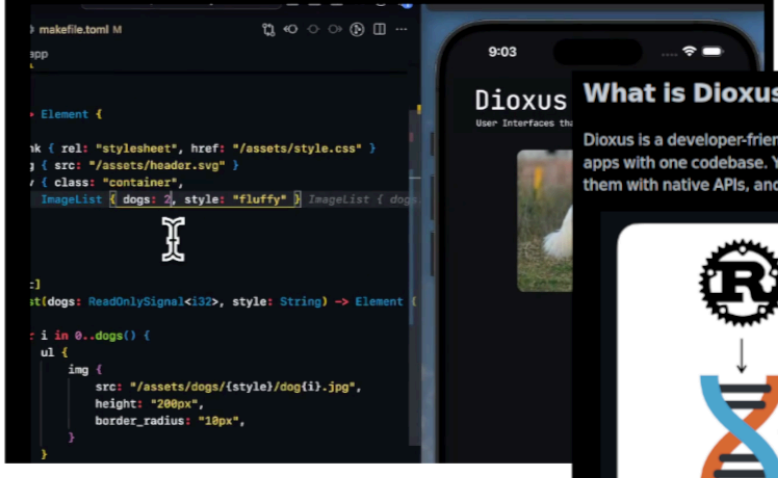
Dioxus is a framework for building cross-platform apps with the Rust programming language. With one codebase, you can build apps that run on web, desktop, and mobile platforms.

src/readme.rs

```rust
use dioxus::prelude::*;

pub fn App() -> Element {
    let mut count = use_signal(|| 0);

    rsx! {
        h1 { "High-Five counter: {count}" }
        button { onclick: move |_| count += 1, "Up high!" }
        button { onclick: move |_| count -= 1, "Down low!" }
    }
}
```

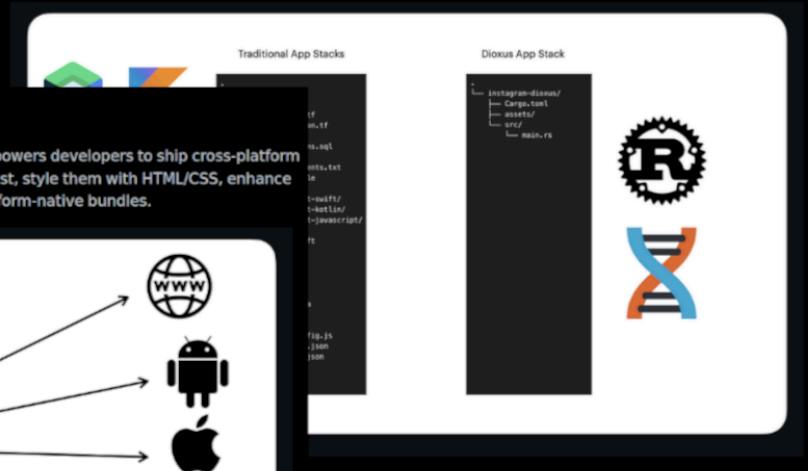
Stellar Developer Experience

With Dioxus, we try to maintain a very high bar for developer experience. We believe that building apps should be fun and straightforward. We've worked to push forward Rust itself, developing technologies like Subsecond Rust hot-reloading, WASM bundle-splitting, linker-based asset bundling, and a modular WGPU-based HTML/CSS renderer.



Why Dioxus?

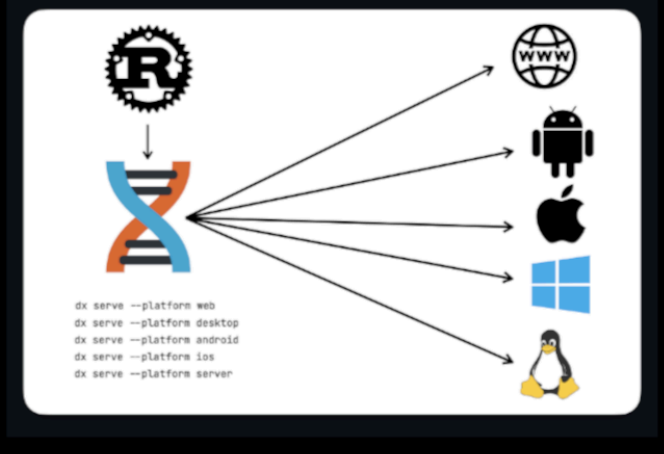
We built Dioxus because we believe the current standard of building apps is too complex. Developers need to learn and install dozens of different tools just to get their app into the world. We need a simpler and more powerful framework to bring apps from ideas to reality.



k that is fast, flexible, and has a minimal learning curve.
launch their app from idea to production as fast as
ls and a simpler architecture make it easier to develop
ship faster and are more likely to succeed.

What is Dioxus?

Dioxus is a developer-friendly framework that empowers developers to ship cross-platform apps with one codebase. You write your apps in Rust, style them with HTML/CSS, enhance them with native APIs, and distribute them as platform-native bundles.



In many ways, Dioxus is similar to Flutter: we integrate our own build tools, foster an ecosystem, and provide a GUI framework. In key areas, Dioxus takes a different approach:

- Apps are declared with HTML and CSS instead of a custom styling solution
- Reactivity is inspired by web frameworks like React and SolidJS
- Dioxus code runs natively with no virtual machine, enabling zero-overhead calls to system APIs

Vous venez plutôt de Rails et RoR ? No problemo…

Docs Blog Casts

Build apps locally and save lots

No need for SaaS or cloud services. Save time, money, and effort
auth, workers, emails & more out of the box.

```
// Literals
format::text("Loco")

// Tera view engine
format::render().view(v, "home/hello.html", json!({}))

// strongly typed JSON responsed, backed by `serde`
format::json(Health { ok: true })

// Etags, cookies, and more
format::render().etag("loco-etag")?.empty()
```

Views

Use server-rendered templates with Tera or JSON. Loco can render views on the server or work with a frontend app seamlessly. Configure your fullstack set up any way you like.

```
impl worker::Worker<DownloadArgs> for UsersReportWorker {
    async fn perform(&self, args: DownloadArgs) -> worker::Result<()> {
        let all = Users::find()
            .all(&self.ctx.db)
            .await
            .map_err(Box::from)?;
        for user in &all {
            println!("user: {}", user.id);
        }
        Ok(())
    }
}
```

Background Jobs

Perform compute or I/O intensive jobs in the background with a Redis backed queue, or with threads. Implementing a worker is as simple as implementing a `**perform**` function for the `**Worker**` trait.

```
impl Model {
    pub async fn find_by_email(db: &DatabaseConnection, email: &str)
    -> ModelResult<Self> {

        Users::find()
            .filter(eq(Column::Email, email))
            .one(db).await?
            .ok_or_else(|| ModelError::EntityNotFound)
    }

    pub async fn create_report(&self, ctx: &AppContext) -> Result<()> {
        ReportWorker::perform_later(
            &ctx,
            ReportArgs{ user_id: self.id }
        ).await?;
    }
}
```

Models

Model your business with rich entities and avoid writing SQL, backed by SeaORM. Build relations, validation and custom logic on your entities for the best maintainability.

```
$ cargo loco generate deployment
› Choose your deployment ›
pub async fn get_one(
    respond_to: RespondTo,
    Path(id): Path<i32>,
    State(ctx): State<AppContext>,
) -> Result<Response> {
    let item = Notes::find_by_id(id).one(&ctx.db).await?;
    match respond_to {
        RespondTo::Html => html_view(&item),
        _ => format::json(item),
    }
}

pub fn routes() -> Routes {
    Routes::new()
        .prefix("notes")
        .add("/{id}", get(get_one))
}
```

Deployment

configurations with a guided CLI interface.
options for tailored deployment setups.

Controllers

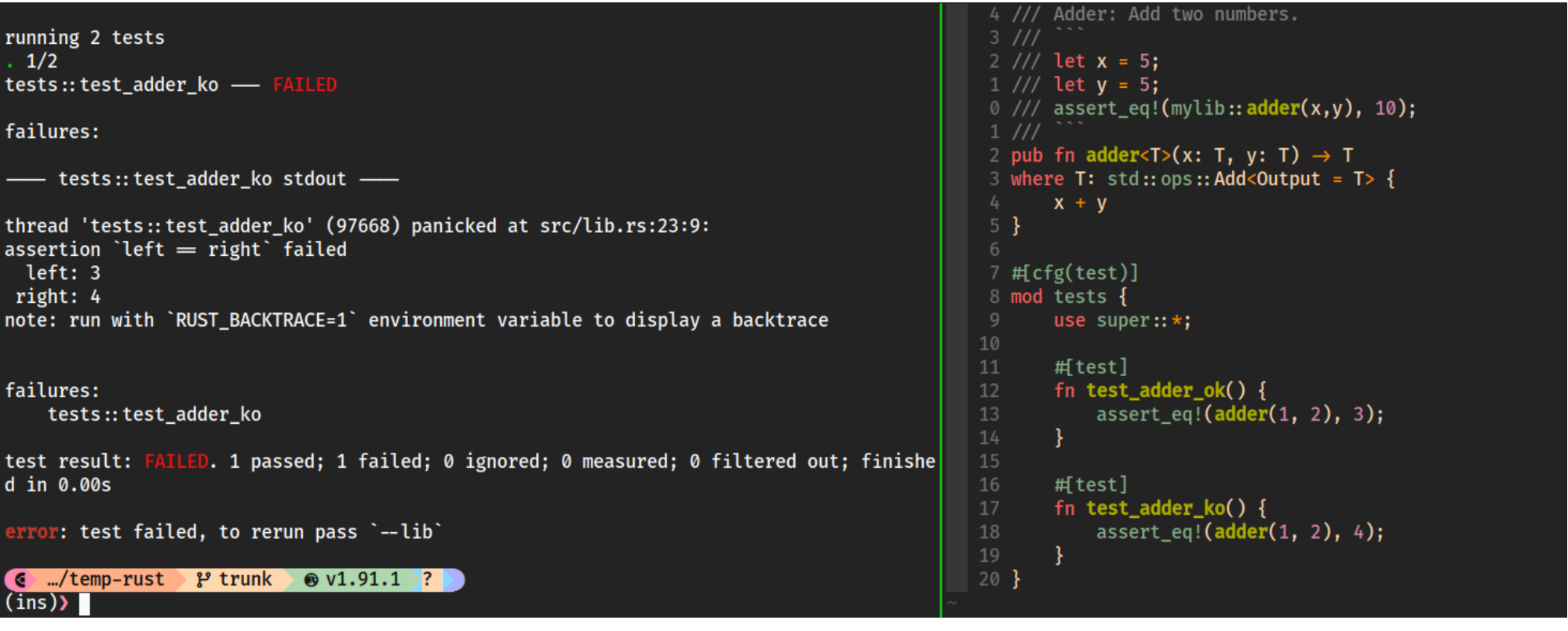
Handle Web requests parameters, body, validation, and render a response that is content-aware. We use Axum for the best performance, simplicity and extensibility.

```
jobs:
  db_vaccum:
    run: "db_vaccum.sh"
    shell: true
    schedule: "0 0 * * *"
    tags: ["maintenance"]

  send_birthday:
    run: "user_birthday_task"
    schedule: "Run every 2 hours"
    tags: ["marketing"]
```

Scheduler

Simplifies the traditional, often cumbersome crontab system, making it easier and more elegant to schedule tasks or shell scripts.

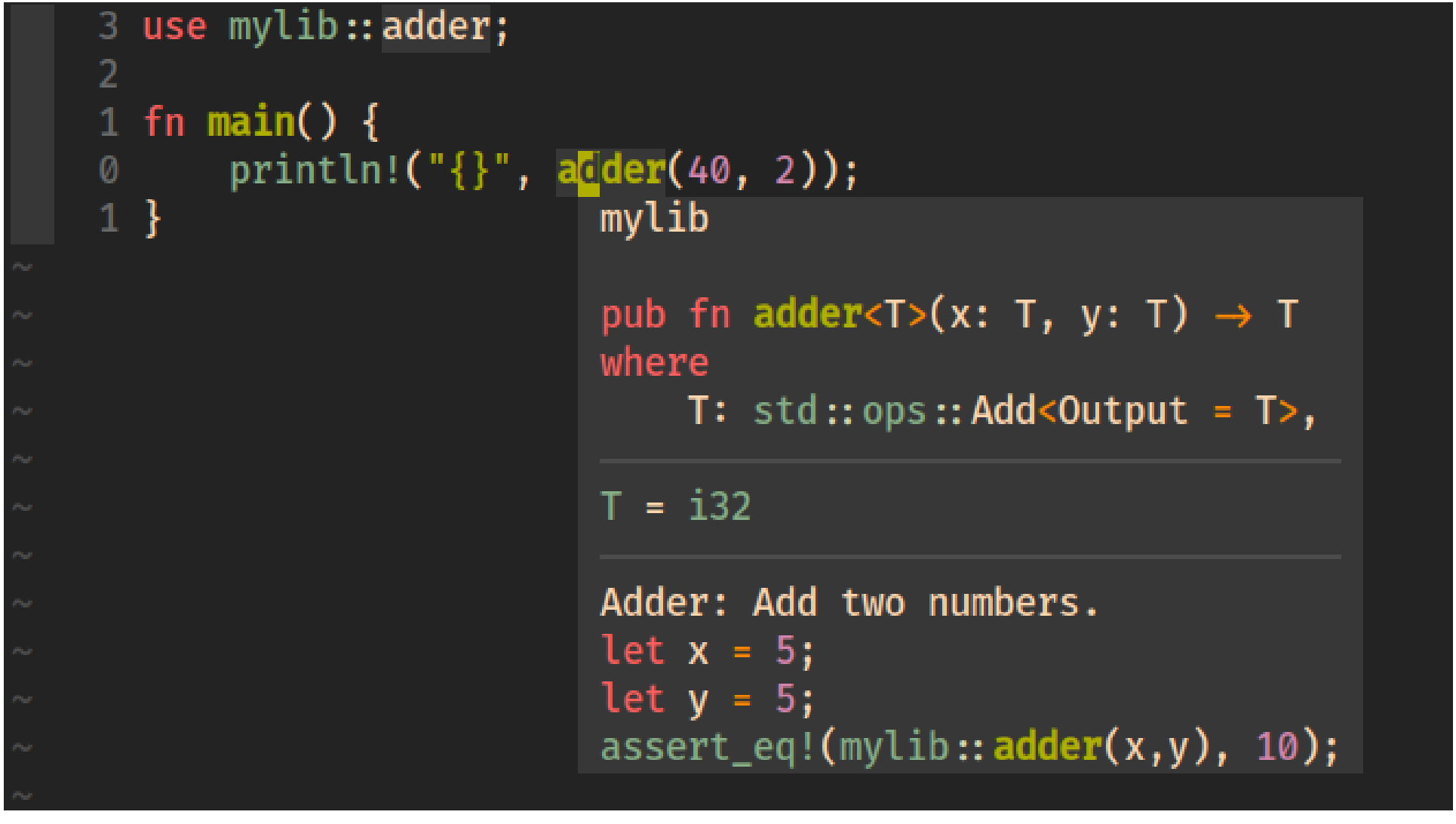
Une manière de tester les applications qui n'a pas son pareil

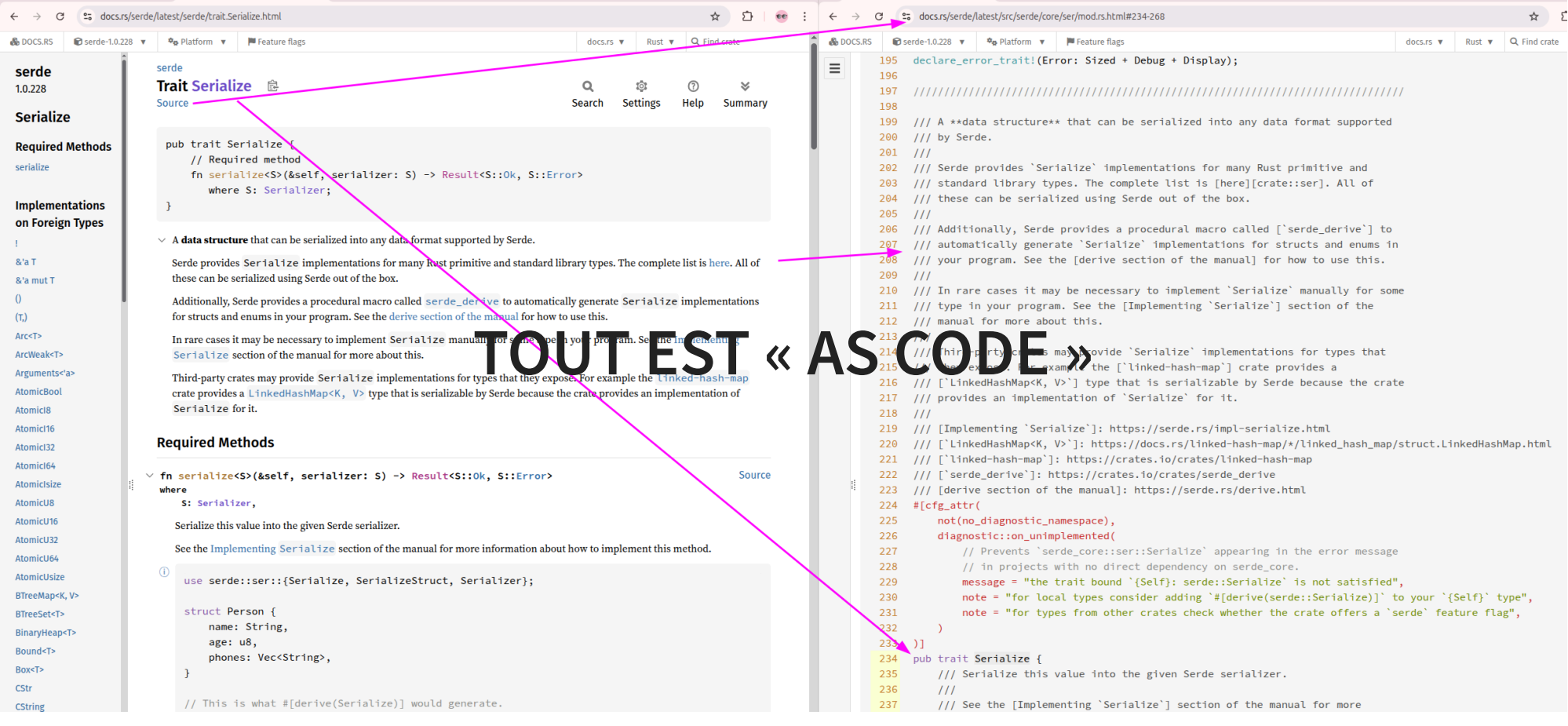


```
running 2 tests
. 1/2
tests::test_adder_ko ── FAILED

failures:

─── tests::test_adder_ko stdout ───

thread 'tests::test_adder_ko' (97668) panicked at src/lib.rs:23:9:
assertion `left == right` failed
  left: 3
 right: 4
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::test_adder_ko

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finishe
d in 0.00s

error: test failed, to rerun pass `--lib`

 …/temp-rust  trunk  v1.91.1 ?
(ins)❯
```

```rust
/// Adder: Add two numbers.
///
/// ```
/// let x = 5;
/// let y = 5;
/// assert_eq!(mylib::adder(x,y), 10);
/// ```
pub fn adder<T>(x: T, y: T) -> T
where T: std::ops::Add<Output = T> {
    x + y
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_adder_ok() {
        assert_eq!(adder(1, 2), 3);
    }

    #[test]
    fn test_adder_ko() {
        assert_eq!(adder(1, 2), 4);
    }
}
```

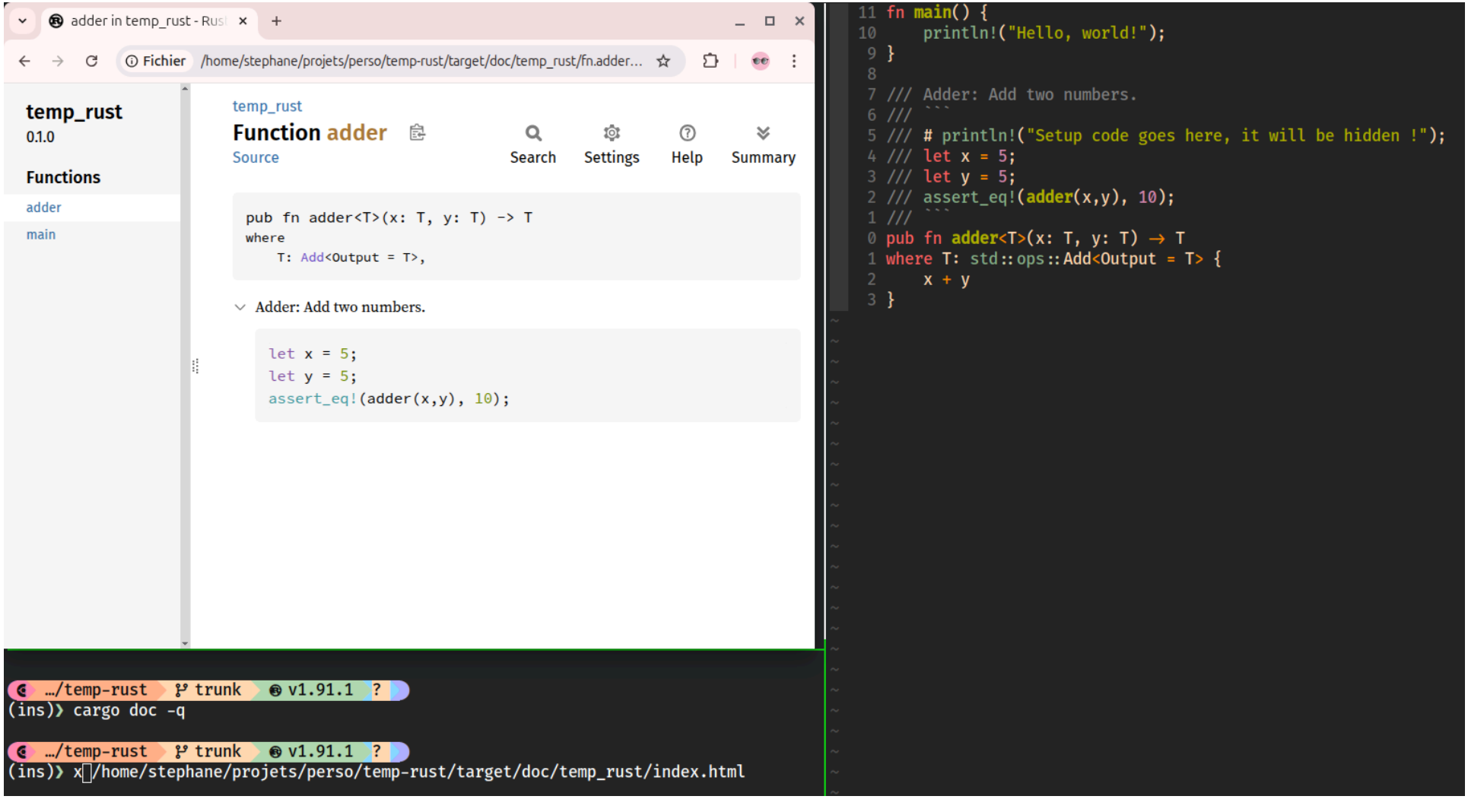
La doc des projets Rust est… *chef kiss* 👌


use mylib::adder;
fn main() {
println!("{}", adder(40, 2));
}
mylib
pub fn adder<T>(x: T, y: T) → T
where
T: std::ops::Add<Output = T>,
T = i32
Adder: Add two numbers.
let x = 5;
let y = 5;
assert_eq!(mylib::adder(x,y), 10);

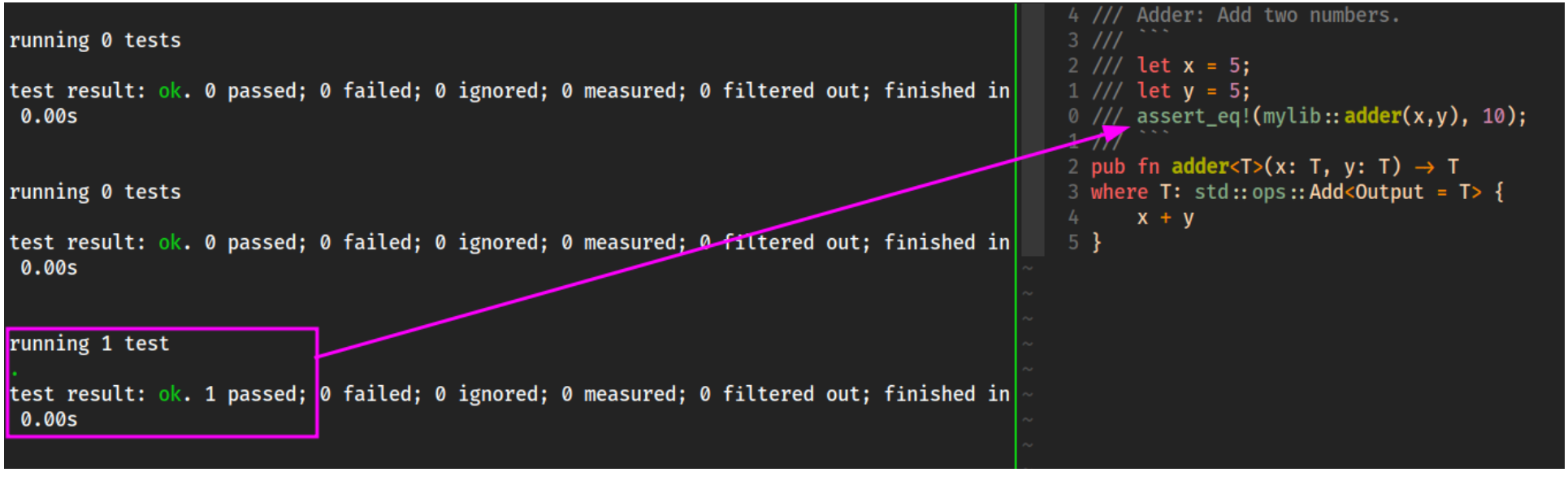


TOUT EST « AS CODE »

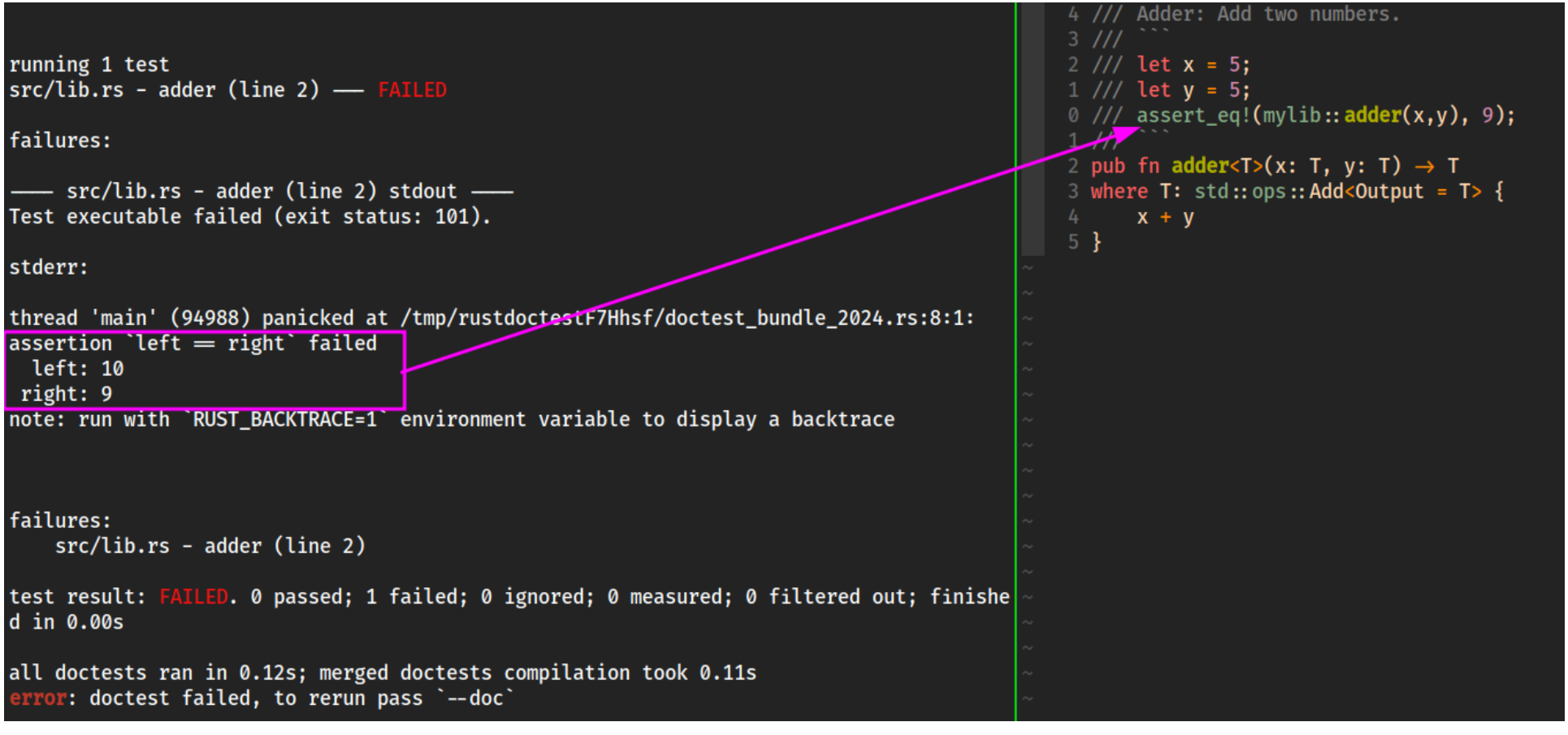
`cargo doc`, tout simplement



Oh, et ils sont *par défaut* testé par `cargo test` 😎

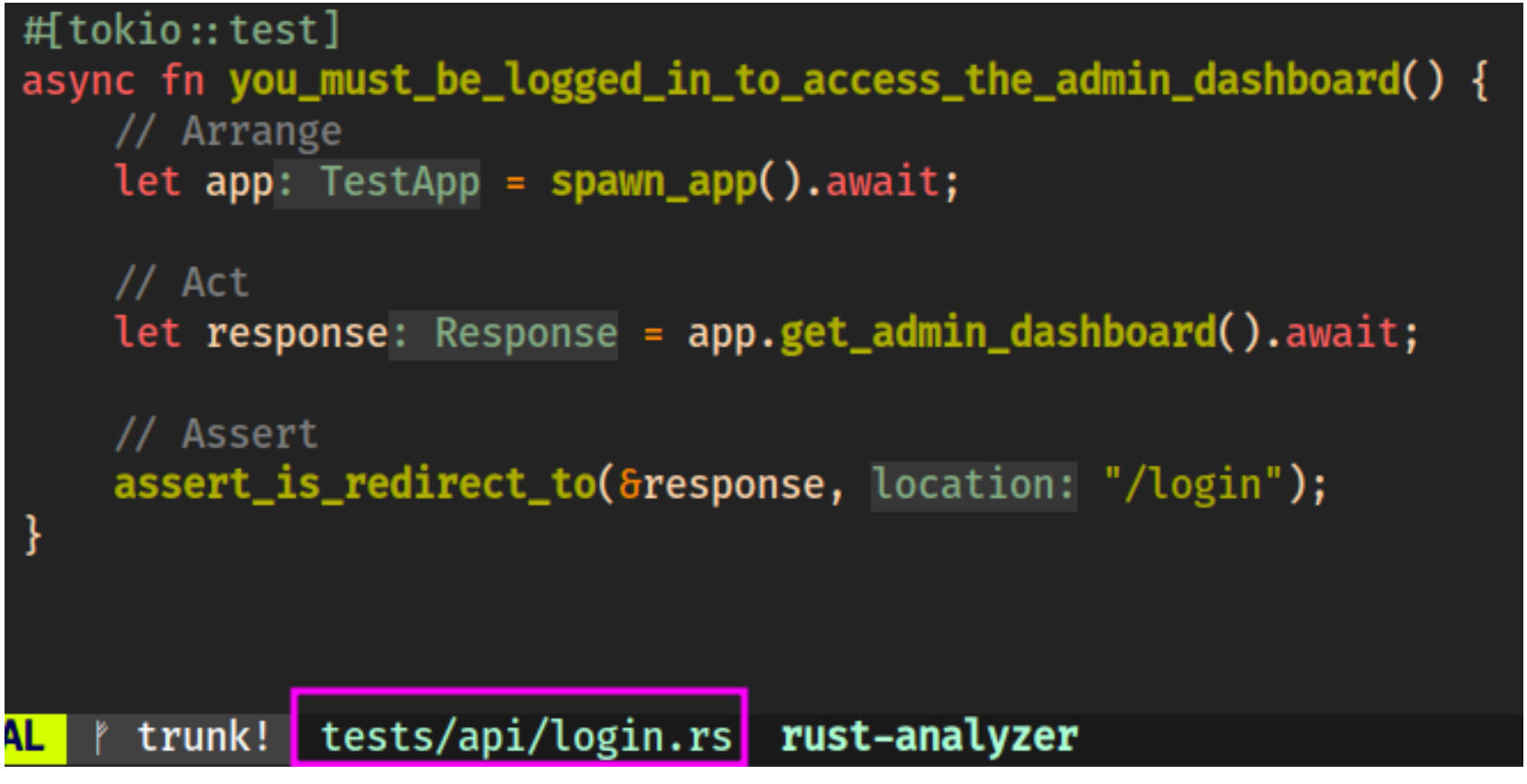


Votre doc contribue ne fait pas que décrire vos invariants, elle les contrôle ! 💪



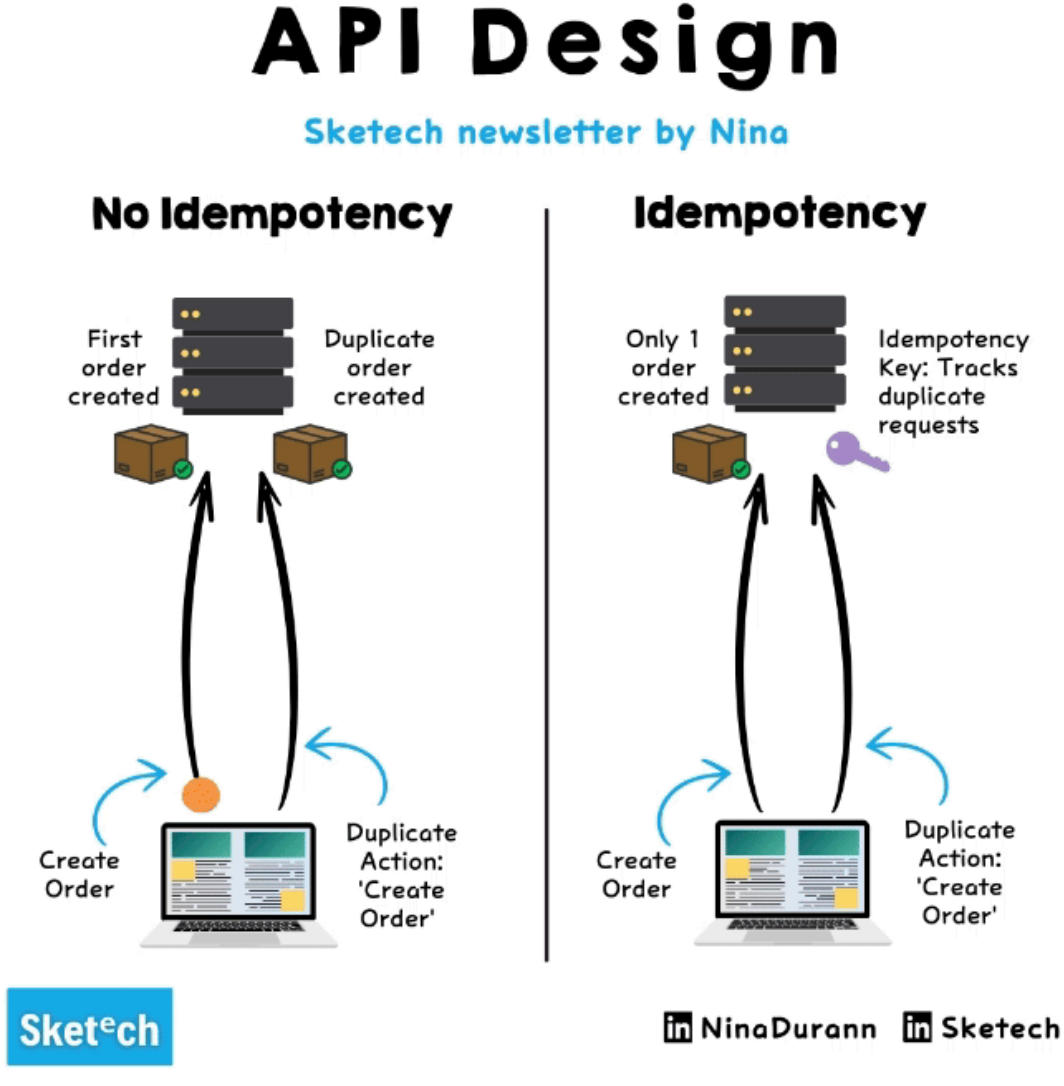
Tests d'Intégration ?

Facile ! On les met dans `/tests` (avec du code de setup si besoin)



D'ailleurs… Il faut qu'on parle de Wiremocks 🤩

Un cas difficile à tester : L'idempotence



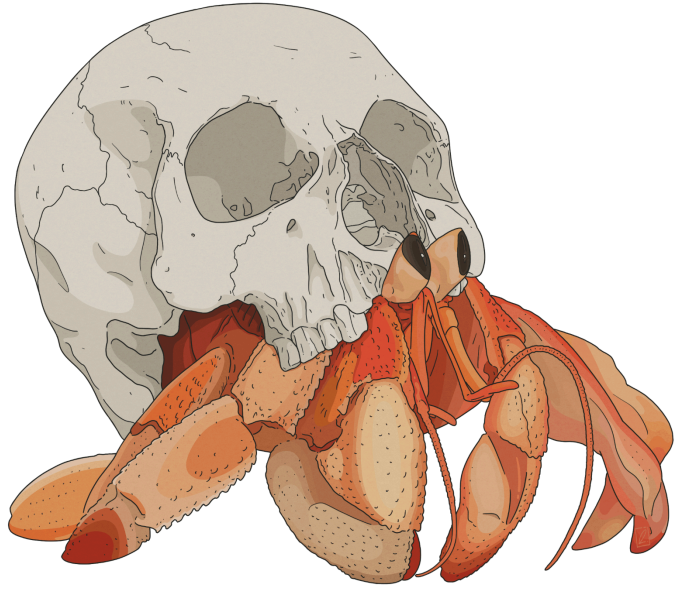
```rust
#[tokio::test]
async fn newsletter_creation_is_idempotent() {
    // Arrange
    let app: TestApp = spawn_app().await;
    create_confirmed_subscriber(&app).await;

    Mock::given(matcher: path("/email")) MockBuilder
        .and(matcher: method("POST")) MockBuilder
        .respond_with(responder: ResponseTemplate::new(200)) Mock
        // Assert made at Drop
        .expect(1) Mock
        .mount(&app.email_server) impl Future<Output = ()>
        .await;

    // Act - Part 1 - Login
    app.test_user.login(&app).await;

    // Act - Part 2 - Post new issue
+-- 16 lines: let id = Uuid::new_v4().to_string();·····

    // Act - Part 3 - Post duplicate request
+--  9 lines: assert_is_redirect_to(·····

    // Assert made at Drop (See Mock)
} async fn newsletter_creation_is_idempotent
```

Si vous voulez aller plus loin...

(parce qu'il y en a encore plein, des dingueries comme ça 😅, dans Rust)

ZERO TO PRODUCTION IN RUST

AN OPINIONATED INTRODUCTION TO BACKEND DEVELOPMENT



LUCA PALMIERI

Learn Rust With Entirely Too Many Linked Lists

I fairly frequently get asked how to implement a linked list in Rust. The answer honestly depends on what your requirements are, and it's obviously not super easy to answer the question on the spot. As such I've decided to write this book to comprehensively answer the question once and for all.

In this series I will teach you basic and advanced Rust programming entirely by having you implement 6 linked lists. In doing so, you should learn:

- The following pointer types: `&`, `&mut`, `Box`, `Rc`, `Arc`, `*const`, `*mut`, `NonNull` (?)
- Ownership, borrowing, inherited mutability, interior mutability, Copy
- All The Keywords: struct, enum, fn, pub, impl, use, ...
- Pattern matching, generics, destructors
- Testing, installing new toolchains, using `miri`
- Unsafe Rust: raw pointers, aliasing, stacked borrows, UnsafeCell, variance

Yes, linked lists are so truly awful that you deal with all of these concepts in making them real.

Everything's in the sidebar (may be collapsed on mobile), but for quick reference, here's what we're going to be making:

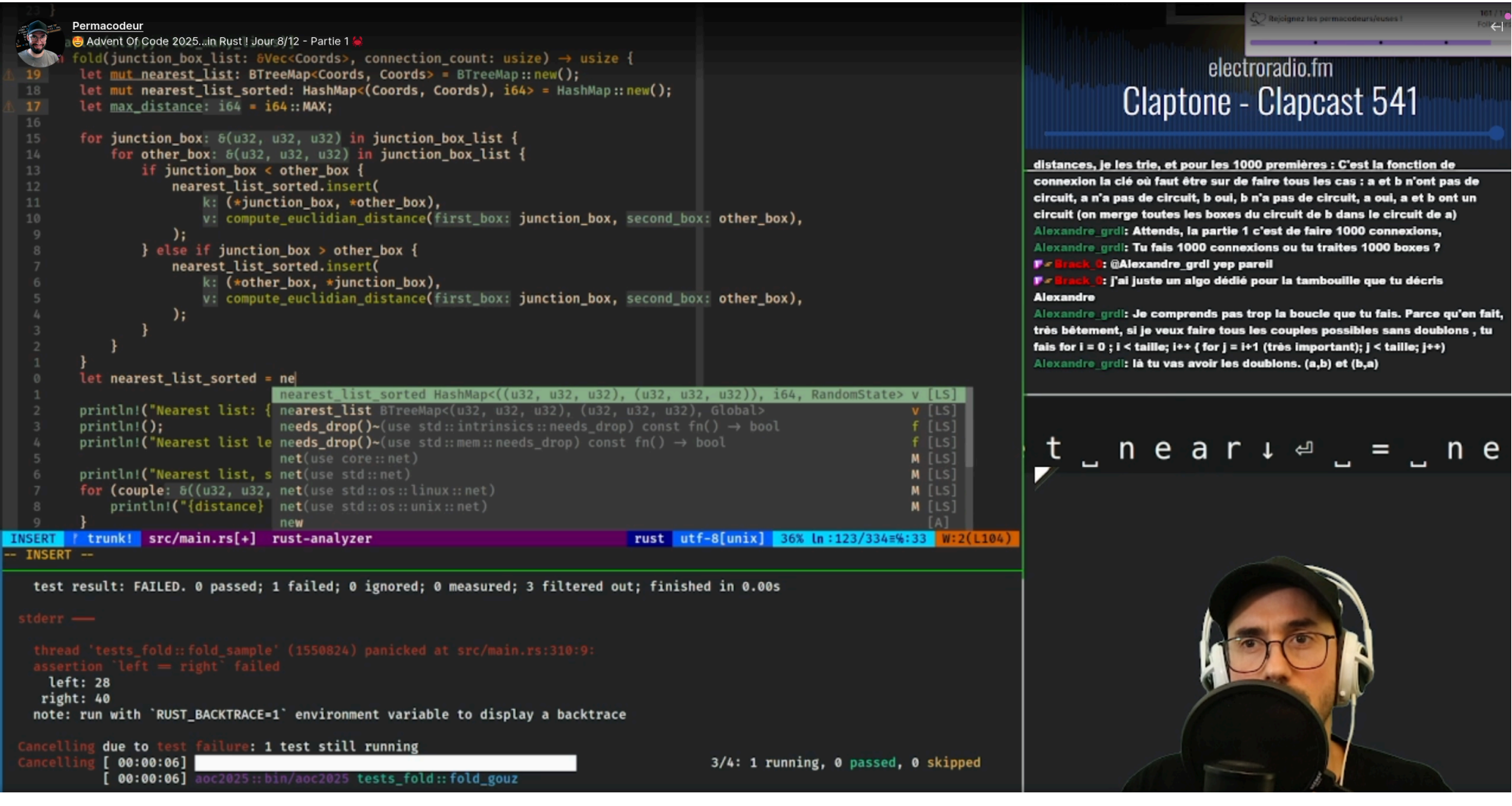
1. A Bad Singly-Linked Stack
2. An Ok Singly-Linked Stack
3. A Persistent Singly-Linked Stack
4. A Bad But Safe Doubly-Linked Deque
5. An Unsafe Singly-Linked Queue
6. TODO: An Ok Unsafe Doubly-Linked Deque
7. Bonus: A Bunch of Silly Lists

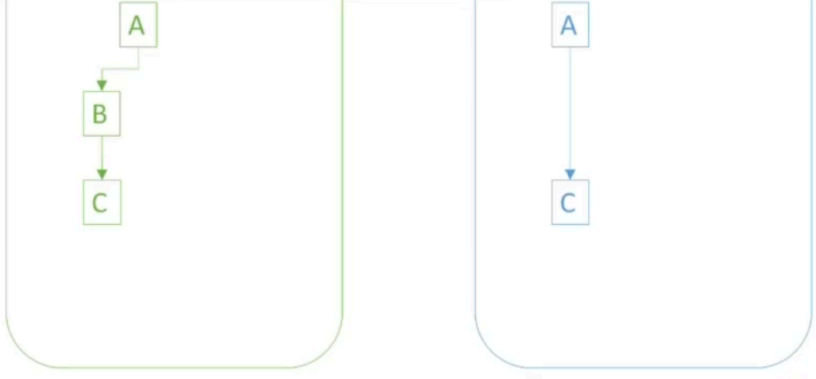
Qui suis-je ? 🫣

👔

🌐

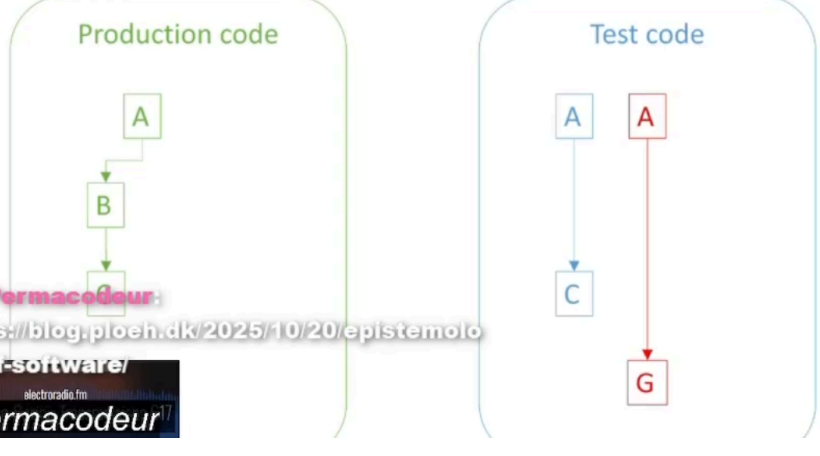
🦀


Permacodeur
Advent Of Code 2025...in Rust ! Jour 8/12 - Partie 1
fold(junction_box_list: &Vec<Coords>, connection_count: usize) -> usize {
let mut nearest_list: BTreeMap<Coords, Coords> = BTreeMap::new();
let mut nearest_list_sorted: HashMap<(Coords, Coords), i64> = HashMap::new();
let max_distance: i64 = i64::MAX;
for junction_box: &(u32, u32, u32) in junction_box_list {
for other_box: &(u32, u32, u32) in junction_box_list {
if junction_box < other_box {
nearest_list_sorted.insert(
k: (*junction_box, *other_box),
v: compute_euclidian_distance(first_box: junction_box, second_box: other_box),
);
} else if junction_box > other_box {
nearest_list_sorted.insert(
k: (*other_box, *junction_box),
v: compute_euclidian_distance(first_box: junction_box, second_box: other_box),
);
}
}
}
let nearest_list_sorted = ne
println!("Nearest list: {
println!();
println!("Nearest list le
println!("Nearest list, s
for (couple: &((u32, u32,
println!("{distance}
}
nearest_list_sorted HashMap<((u32, u32, u32), (u32, u32, u32)), i64, RandomState> v [LS]
nearest_list BTreeMap<u32, u32, u32), (u32, u32, u32), Global> v [LS]
needs_drop()~(use std::intrinsics::needs_drop) const fn() -> bool f [LS]
needs_drop()~(use std::mem::needs_drop) const fn() -> bool f [LS]
net(use core::net) M [LS]
net(use std::net) M [LS]
net(use std::os::linux::net) M [LS]
net(use std::os::unix::net) M [LS]
new [A]
INSERT trunk! src/main.rs[+] rust-analyzer rust utf-8[unix] 36% ln:123/334≡%:33 W:2(L104)
-- INSERT --
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 3 filtered out; finished in 0.00s
stderr ———
thread 'tests_fold::fold_sample' (1550824) panicked at src/main.rs:310:9:
assertion `left == right` failed
left: 28
right: 40
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Cancelling due to test failure: 1 test still running
Cancelling [00:00:06]
[00:00:06] aoc2025::bin/aoc2025 tests_fold::fold_gouz
3/4: 1 running, 0 passed, 0 skipped
Rejoignez les permacodeurs/euses !
electroradio.fm
Claptone - Clapcast 541
distances, je les trie, et pour les 1000 premières : C'est la fonction de connexion la clé où faut être sur de faire tous les cas : a et b n'ont pas de circuit, a n'a pas de circuit, b oui, b n'a pas de circuit, a oui, a et b ont un circuit (on merge toutes les boxes du circuit de b dans le circuit de a)
Alexandre_grdl: Attends, la partie 1 c'est de faire 1000 connexions,
Alexandre_grdl: Tu fais 1000 connexions ou tu traites 1000 boxes ?
Brack_0: @Alexandre_grdl yep pareil
Brack_0: j'ai juste un algo dédié pour la tambouille que tu décris Alexandre
Alexandre_grdl: Je comprends pas trop la boucle que tu fais. Parce qu'en fait, très bêtement, si je veux faire tous les couples possibles sans doublons , tu fais for i = 0 ; i < taille; i++ { for j = i+1 (très important); j < taille; j++)
Alexandre_grdl: là tu vas avoir les doublons. (a,b) et (b,a)
t _ n e a r ↓ ⏎ _ = _ n e



This, again, is a falsifiable prediction. If, despite expectations, the test fails, you know that something is wrong. Most likely, it's the implementation that you just wrote, but it could also be the test which, after all, is somehow defective. Or perhaps a circuit in your computer was struck by a cosmic ray. On the other hand, if the test passes, you've failed to falsify your prediction, which is the best you can hope for.

You now write a second test, which comes with the implicit falsifiable prediction: *If I run all tests, the new test will fail*.



Les liens

Vos feedbacks

@StephaneTrebel

Rust 🦀 a désormais tout ce qu'il faut pour délivrer un MAX de valeur.

Jetez-y un œil ! 👀

Les liens

Vos feedbacks

@StephaneTrebel

Rust 🦀 a désormais tout ce qu'il faut pour vous permettre de délivrer de la valeur. Aucune raison de ne pas s'y mettre ! 😉