

An Introduction To Property-Based Testing In Rust

January 03, 2021 · 3912 words · 20 min

This article is a sample from **Zero To Production In Rust**, a hands-on introduction to backend development in Rust.

You can get a copy of the book at zero2prod.com.

TL;DR

We need to verify a user-provided email address. We'll use this as an opportunity to explore the tools available in the Rust ecosystem to perform property-based testing.

We will start with fake, a Rust crate to generate randomised test data.

We will then combine fake with quickcheck to generate multiple samples on every cargo test invocation using its Arbitrary trait.

Chapter 6.5

1. Previously In Zero To Production In Rust
2. The Email Format
3. The `SubscriberEmail` Type
 - Breaking The Domain Sub-Module
 - Skeleton Of A New Type
4. Property-based Testing
 - How To Generate Random Test Data With `fake`
 - `quickcheck` Vs `proptest`
 - Getting Started With `quickcheck`
 - Implementing The `Arbitrary` Trait
5. Payload Validation
 - Refactoring With `TryFrom`
6. Summary

Previously In Zero To Production In Rust



You can find the snapshot of the codebase at the beginning of this chapter on [GitHub](#).

At the beginning of [chapter six](#) we wrote an integration test to stress how poor our user input validation was - let's look at it again:

```
#!/ tests/health_check.rs
#![ [ ... ]

#[tokio::test]
async fn subscribe_returns_a_400_when_fields_are_present_but_invalid() {
    // Arrange
    let app = spawn_app().await;
    let client = request::Client::new();
    let test_cases = vec![
        ("name=&email=ursula_le_guin%40gmail.com", "empty name"),
        ("name=Ursula&email=", "empty email"),
        ("name=Ursula&email=definitely-not-an-email", "invalid email"),
    ];

    for (body, description) in test_cases {
        // Act
        let response = client
            .post(&format!("{}",/subscriptions", &app.address))
            .header("Content-Type", "application/x-www-form-urlencoded")
            .body(body)
            .send()
            .await
            .expect("Failed to execute request.");

        // Assert
        assert_eq!(
            400,
            response.status().as_u16(),
            "The API did not return a 400 Bad Request when the payload was {}.\"",
```

```
        description
    );
}
}
```



We have then worked to introduce input validation into our newsletter project: the subscriber name in the payload of our `POST /subscriptions` endpoint is now thoroughly inspected before any saving or processing takes place.

That is how we got the `empty name` case to pass, but we are still failing on the `empty email` one:

```
--- subscribe_returns_a_400_when_fields_are_present_but_invalid stdout ----
thread 'subscribe_returns_a_400_when_fields_are_present_but_invalid'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `200`:
The API did not return a 400 Bad Request when the payload was empty email.',
tests/health_check.rs:167:9
```

Turning this test green will be our focus for today.

The Email Format

We are all intuitively familiar with the *common* structure of an email address - `xxx@yyy.zzz` - but the subject quickly gets more complicated if you desire to be rigorous and avoid bouncing email addresses that are actually valid.

How do we establish if an email address is “valid”?

There are a few Request For Comments (RFC) by the Internet Engineering Task Force (IETF) outlining the expected structure of an email address - [RFC 6854](#), [RFC 5322](#), [RFC 2822](#). We would have to read them, digest the material and then come up with an `is_valid_email` function that matches the specification.

Unless you have a keen interest in understanding the subtle nuances of the email address format, I would suggest you to take a step back: it is quite messy. So messy that even the HTML specification is *willfully* non-compliant with the RFCs we just linked.

Our best shot is to look for an existing library that has stared long and hard at the problem to provide us with a plug-and-play solution. Luckily enough, there is at least one

in the Rust ecosystem - the `validator` crate¹



The `SubscriberEmail` Type

We will follow the same strategy we used for name validation - encode our invariant ("this string represents a valid email") in a new `SubscriberEmail` type.

Breaking The Domain Sub-Module

Before we get started though, let's make some space - let's break our `domain` sub-module (`domain.rs`) into multiple smaller files, one for each type, similarly to what we did for routes back in Chapter 3. Our current folder structure (under `src`) is:

```
src/  
  routes/  
    [ ... ]  
  domain.rs  
  [ ... ]
```

We want to have

```
src/  
  routes/  
    [ ... ]  
  domain/  
    mod.rs  
    subscriber_name.rs  
    subscriber_email.rs  
    new_subscriber.rs  
  [ ... ]
```

Unit tests should be in the same file of the type they refer to. We will end up with:

```
//! src/domain/mod.rs  
  
mod subscriber_name;  
mod subscriber_email;
```

```
mod new_subscriber;
```

```
pub use subscriber_name::SubscriberName;
```

```
pub use new_subscriber::NewSubscriber;
```

```
//! src/domain/subscriber_name.rs
```

```
use unicode_segmentation::UnicodeSegmentation;
```

```
#[derive(Debug)]
```

```
pub struct SubscriberName(String);
```

```
impl SubscriberName {
```

```
    // [ ... ]
```

```
}
```

```
impl AsRef<str> for SubscriberName {
```

```
    // [ ... ]
```

```
}
```

```
#[cfg(test)]
```

```
mod tests {
```

```
    // [ ... ]
```

```
}
```

```
//! src/domain/subscriber_email.rs
```

```
// Still empty, ready for us to get started!
```

```
//! src/domain/new_subscriber.rs
```

```
use crate::domain::subscriber_name::SubscriberName;
```

```
pub struct NewSubscriber {
```

```
    pub email: String,
```

```
    pub name: SubscriberName,
```

```
}
```

No changes should be required to other files in our project - the API of our module has not changed thanks to our `pub use` statements in `mod.rs`.

Skeleton Of A New Type

Let's add a barebone `SubscriberEmail` type: no validation, just a wrapper around a `String` and a convenient `AsRef` implementation:

```
//! src/domain/subscriber_email.rs

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

impl SubscriberEmail {
    pub fn parse(s: String) -> Result<SubscriberEmail, String> {
        // TODO: add validation!
        Ok(Self(s))
    }
}

impl AsRef<str> for SubscriberEmail {
    fn as_ref(&self) -> &str {
        &self.0
    }
}
```

```
//! src/domain/mod.rs
```

```
mod new_subscriber;
mod subscriber_email;
mod subscriber_name;

pub use new_subscriber::NewSubscriber;
pub use subscriber_email::SubscriberEmail;
pub use subscriber_name::SubscriberName;
```



We start with tests this time: let's come up with a few examples of invalid emails that should be rejected.

```
//! src/domain/subscriber_email.rs
```

```
#[derive(Debug)]
pub struct SubscriberEmail(String);

// [ ... ]

#[cfg(test)]
mod tests {
    use super::SubscriberEmail;
    use claim::assert_err;

    #[test]
    fn empty_string_is_rejected() {
        let email = "".to_string();
        assert_err!(SubscriberEmail::parse(email));
    }

    #[test]
    fn email_missing_at_symbol_is_rejected() {
        let email = "ursuladomain.com".to_string();
        assert_err!(SubscriberEmail::parse(email));
    }
}
```

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



Running `cargo test domain` confirms that all test cases are failing:

```
failures:
    domain::subscriber_email::tests::email_missing_at_symbol_is_rejected
    domain::subscriber_email::tests::email_missing_subject_is_rejected
    domain::subscriber_email::tests::empty_string_is_rejected

test result: FAILED. 6 passed; 3 failed; 0 ignored; 0 measured; 0 filtered out
```

Time to bring `validator` in:

```
#! Cargo.toml
# [ ... ]
[dependencies]
validator = "0.14"
# [ ... ]
```

Our `parse` method will just delegate all the heavy-lifting to `validator::validate_email`:

```
//! src/domain/subscriber_email.rs

use validator::validate_email;

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

impl SubscriberEmail {
    pub fn parse(s: String) -> Result<SubscriberEmail, String> {
        if validate_email(&s) {
            Ok(Self(s))
        }
    }
}
```



```
    } else {
        Err(format!("{}", s) is not a valid subscriber email.", s))
    }
}

// [ ... ]
```



As simple as that - all our tests are green now!

There is a caveat - all our tests cases are checking for *invalid* emails. We should also have at least one test checking that valid emails are going through.

We could hard-code a known valid email address in a test and check that it is parsed successfully - e.g. `ursula@domain.com`.

What value would we get from that test case though? It would only re-assure us that *a specific email address* is correctly parsed as valid.

Property-based Testing

We could use another approach to test our parsing logic: instead of verifying that a certain set of inputs is correctly parsed, we could build a random generator that produces valid values and check that our parser does not reject them.

In other words, we verify that our implementation displays a certain *property* - "No valid email address is rejected".

This approach is often referred to as *property-based testing*.

If we were working with time, for example, we could *repeatedly* sample three random integers

- H, between 0 and 23 (inclusive);
- M, between 0 and 59 (inclusive);
- S, between 0 and 59 (inclusive);

and verify that `H:M:S` is always correctly parsed.

Property-based testing significantly increases the range of inputs that we are validating, and therefore our confidence in the correctness of our code, but it does not *prove* that our parser is correct - it does not *exhaustively* explore the input space (except for tiny ones).

Let's see what property testing would look like for our `SubscriberEmail`.



How To Generate Random Test Data With `fake`

First and foremost, we need a random generator of valid emails.

We could write one, but this a great opportunity to introduce the `fake` crate.

`fake` provides generation logic for both primitive data types (integers, floats, strings) and higher-level objects (IP addresses, country codes, etc.) - in particular, emails! Let's add `fake` as a development dependency of our project:

```
# Cargo.toml
# [ ... ]

[dev-dependencies]
# [ ... ]
# We are not using fake >= 2.4 because it relies on rand 0.8
# which has been recently released and it is not yet used by
# quickcheck (solved in its upcoming 1.0 release!)
fake = "~2.3"
```

Let's use it in a new test:

```
//! src/domain/subscriber_email.rs

// [ ... ]

#[cfg(test)]
mod tests {
    // We are importing the `SafeEmail` faker!
    // We also need the `Fake` trait to get access to the
    // `.fake` method on `SafeEmail`
    use fake::faker::internet::en::SafeEmail;
    use fake::Fake;
    // [ ... ]

    #[test]
    fn valid_emails_are_parsed_successfully() {
```

```
    let email = SafeEmail().fake();
    claim::assert_ok!(SubscriberEmail::parse(email));
}
}
```



Every time we run our test suite, `SafeEmail().fake()` generates a new random valid email which we then use to test our parsing logic.

This is already a major improvement compared to a hard-coded valid email, but we would have to run our test suite several times to catch an issue with an edge case. A fast-and-dirty solution would be to add a `for` loop to the test, but, once again, we can use this as an occasion to delve deeper and explore one of the available testing crates designed around property-based testing.

`quickcheck` Vs `proptest`

There are two mainstream options for property-based testing in the Rust ecosystem: `quickcheck` and `proptest`.

Their domains overlap, although each shines in its own niche - check their READMEs for all the nitty gritty details.


For our project we will go with `quickcheck` - it is fairly simple to get started with and it does not use too many macros, which makes for a pleasant IDE experience.

Getting Started With `quickcheck`

Let's have a look at one of their examples to get the gist of how it works:

```
/// The function we want to test.
fn reverse<T: Clone>(xs: &[T]) -> Vec<T> {
    let mut rev = vec!();
    for x in xs.iter() {
        rev.insert(0, x.clone())
    }
    rev
}

#[cfg(test)]
mod tests {
```



```
#[quickcheck_macros::quickcheck]
fn prop(xs: Vec<u32>) -> bool {
    /// A property that is always true, regardless
    /// of the vector we are applying the function to:
    /// reversing it twice should return the original input.
    xs == reverse(&reverse(&xs))
}
}
```

`quickcheck` calls `prop` in a loop with a configurable number of iterations (100 by default): on every iteration, it generates a new `Vec<u32>` and checks that `prop` returned `true`. If `prop` returns `false`, it tries to **shrink** the generated input to the smallest possible failing example (the shortest failing vector) to help us debug what went wrong.

In our case, we'd like to have something along these lines:

```
#[quickcheck_macros::quickcheck]
fn valid_emails_are_parsed_successfully(valid_email: String) -> bool {
    SubscriberEmail::parse(valid_email).is_ok()
}
}
```

Unfortunately, if we ask for a `String` type as input we are going to get all sorts of garbage which will fail validation.

How do we customise the generation routine?

Implementing The `Arbitrary` Trait

Let's go back to the previous example - how does `quickcheck` know how to generate a `Vec<u32>`?

Everything is built on top of `quickcheck`'s `Arbitrary` trait:

```
pub trait Arbitrary: Clone + Send + 'static {
    fn arbitrary<G: Gen>(g: &mut G) -> Self;

    fn shrink(&self) -> Box<dyn Iterator<Item=Self>> {
        empty_shrinker()
    }
}
```



We have two methods:

- `arbitrary`: given a source of randomness (`g`) it returns an instance of the type;
- `shrink`: it returns a sequence of progressively "smaller" instances of the type to help `quickcheck` find the smallest possible failure case.

`Vec<u32>` implements `Arbitrary`, therefore `quickcheck` knows how to generate random `u32` vectors.

We need to create our own type, let's call it `ValidEmailFixture`, and implement `Arbitrary` for it.

If you look at `Arbitrary`'s trait definition, you'll notice that shrinking is optional: there is a default implementation (using `empty_shrinker`) which results in `quickcheck` outputting the first failure encountered, without trying to make it any smaller or nicer. Therefore we only need to provide an implementation of `Arbitrary::arbitrary` for our `ValidEmailFixture`.

Let's add both `quickcheck` and `quickcheck-macros` as development dependencies:

```
#!/ Cargo.toml
# [ ... ]

[dev-dependencies]
# [ ... ]
quickcheck = "0.9.2"
quickcheck_macros = "0.9.1"
```

Then

```
//! src/domain/subscriber_email.rs
// [ ... ]
```



```
#[cfg(test)]
mod tests {
    // We have removed the `assert_ok` import.
    use claim::assert_err;
    // [ ... ]

    // Both `Clone` and `Debug` are required by `quickcheck`
    #[derive(Debug, Clone)]
    struct ValidEmailFixture(pub String);

    impl quickcheck::Arbitrary for ValidEmailFixture {
        fn arbitrary<G: quickcheck::Gen>(g: &mut G) -> Self {
            let email = SafeEmail().fake_with_rng(g);
            Self(email)
        }
    }

    #[quickcheck_macros::quickcheck]
    fn valid_emails_are_parsed_successfully(valid_email: ValidEmailFixture) ->
    bool {
        SubscriberEmail::parse(valid_email.0).is_ok()
    }
}
```

This is an amazing example of the interoperability you gain by sharing key traits across the Rust ecosystem.

How do we get `fake` and `quickcheck` to play nicely together?

In `Arbitrary::arbitrary` we get `g` as input, an argument of type `G`.

`G` is constrained by a trait bound, `G: quickcheck::Gen`, therefore it must implement the `Gen` trait in `quickcheck`, where `Gen` stands for "generator".

How is `Gen` defined?

```
pub trait Gen: RngCore {
    fn size(&self) -> usize;
}
```

Anything that implements `Gen` must also implement the `RngCore` trait from `rand-core`.

Let's examine the `SafeEmail` faker: it implements the `Fake` trait.

`Fake` gives us a `fake` method, which we have already tried out, but it also exposes a `fake_with_rng` method, where "rng" stands for "random number generator".

What does `fake` accept as a valid random number generator?

```
pub trait Fake: Sized {
    // [ ... ]

    fn fake_with_rng<U, R>(&self, rng: &mut R) -> U where
        R: Rng + ?Sized,
        Self: FakeBase<U>;
}
```

You read that right - any type that implements the `Rng` trait from `rand`, which is automatically implemented by all types implementing `RngCore`!

We can just pass `g` from `Arbitrary::arbitrary` as the random number generator for `fake_with_rng` and *everything just works*!

Maybe the maintainers of the two crates are aware of each other, maybe they aren't, but a community-sanctioned set of traits in `rand-core` gives us painless interoperability. Pretty sweet!

You can now run `cargo test domain` - it should come out green, re-assuring us that our email validation check is indeed not overly prescriptive.

If you want to see the random inputs that are being generated, add a `dbg!`

`(&valid_email.0);` statement to the test and run `cargo test valid_emails -- --nocapture` - tens of valid emails should pop up in your terminal!

Payload Validation

If you run `cargo test`, without restricting the set of tests being run to `domain`, you will see that our integration test with invalid data is still red.

```
--- subscribe_returns_a_400_when_fields_are_present_but_invalid stdout ----
thread 'subscribe_returns_a_400_when_fields_are_present_but_invalid'
panicked at 'assertion failed: `(left == right)`
```

```
left: `400`,
right: `200`:
```

The API did not return a 400 Bad Request when the payload was empty email.',
`tests/health_check.rs:167:9`



Let's integrate our shiny `SubscriberEmail` into the application to benefit from its validation in our `/subscriptions` endpoint.

We need to start from `NewSubscriber`:

```
//! src/domain/new_subscriber.rs

use crate::domain::SubscriberName;
use crate::domain::SubscriberEmail;

pub struct NewSubscriber {
    // We are not using `String` anymore!
    pub email: SubscriberEmail,
    pub name: SubscriberName,
}
```

Hell should break loose if you try to compile the project now.

Let's start with the first error reported by `cargo check`:

```
error[E0308]: mismatched types
  --> src/routes/subscriptions.rs:28:16
   |
28 |         email: form.0.email,
   |                   ^^^^^^^^^^^^^
   |                   expected struct `SubscriberEmail`,
   |                   found struct `std::string::String`
```

It is referring to a line in our request handler, `subscribe`:

```
//! src/routes/subscriptions.rs
// [ ... ]

#[tracing::instrument([ ... ])]
```




```
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let name = match SubscriberName::parse(form.0.name) {
        Ok(name) => name,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let new_subscriber = NewSubscriber {
        // We are trying to assign a string to a field of type
SubscriberEmail!
        email: form.0.email,
        name,
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

We need to mimic what we are already doing for the `name` field: first we parse `form.0.email` then we assign the result (if successful) to `NewSubscriber.email`.

```
//! src/routes/subscriptions.rs

// We added `SubscriberEmail`!
use crate::domain::{NewSubscriber, SubscriberEmail, SubscriberName};
// [ ... ]

#[tracing::instrument([ ... ])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let name = match SubscriberName::parse(form.0.name) {
        Ok(name) => name,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let new_subscriber = NewSubscriber {
        email: form.0.email,
        name,
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

```

        };
let email = match SubscriberEmail::parse(form.0.email) {
    Ok(email) => email,
    Err(_) => return HttpResponse::BadRequest().finish(),
};
let new_subscriber = NewSubscriber { email, name };
// [ ... ]
}

```



Time to move to the second error:

```

error[E0308]: mismatched types
--> src/routes/subscriptions.rs:50:9
|
50 |         new_subscriber.email,
|         ^^^^^^^^^^^^^^^^^^^
|         expected `&str`,
|         found struct `SubscriberEmail`

```

This is in our `insert_subscriber` function, where we perform a SQL INSERT query to store the details of the new subscriber:

```

//! src/routes/subscriptions.rs

// [ ... ]

#[tracing::instrument([ ... ])]
pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
        INSERT INTO subscriptions (id, email, name, subscribed_at)
        VALUES ($1, $2, $3, $4)
        "#,
        Uuid::new_v4(),
    )
    .execute(pool)
    .await
    .map_err(sqlx::Error::from)
}

```



```
// It expects a `&str` but we are passing it
// a `SubscriberEmail` value
new_subscriber.email,
new_subscriber.name.as_ref(),
Utc::now()
)
.execute(pool)
.await
.map_err(|e| {
    tracing::error!("Failed to execute query: {:?}", e);
    e
})?;
Ok(())
}
```

The solution is right there, on the line below - we just need to borrow the inner field of `SubscriberEmail` as a string slice using our implementation of `AsRef<str>`.

```
//! src/routes/subscriptions.rs

// [ ... ]

#[tracing::instrument([ ... ])]
pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
        INSERT INTO subscriptions (id, email, name, subscribed_at)
        VALUES ($1, $2, $3, $4)
        "#,
        Uuid::new_v4(),
        // Using `as_ref` now!
        new_subscriber.email.as_ref(),
        new_subscriber.name.as_ref(),
        Utc::now()
    )
```

```

    )
    .execute(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(())
}

```



That's it - it compiles now!

What about our integration test?

```
cargo test
```

```

running 4 tests
test subscribe_returns_a_400_when_data_is_missing ... ok
test health_check_works ... ok
test subscribe_returns_a_400_when_fields_are_present_but_invalid ... ok
test subscribe_returns_a_200_for_valid_form_data ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

```

All green! We made it!

Refactoring With `TryFrom`


Before we move on let's spend a few paragraphs to refactor the code we just wrote. I am referring to our request handler, `subscribe`:

```

//! src/routes/subscriptions.rs
// [ ... ]

#[tracing::instrument([ ... ])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,

```



```

) -> HttpResponse {
    let name = match SubscriberName::parse(form.0.name) {
        Ok(name) => name,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let email = match SubscriberEmail::parse(form.0.email) {
        Ok(email) => email,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let new_subscriber = NewSubscriber { email, name };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}

```

We can extract the first two statements in a `parse_subscriber` function:

```

//! src/routes/subscriptions.rs
// [ ... ]

pub fn parse_subscriber(form: FormData) -> Result<NewSubscriber, String> {
    let name = SubscriberName::parse(form.name)?;
    let email = SubscriberEmail::parse(form.email)?;
    Ok(NewSubscriber { email, name })
}

#[tracing::instrument([ ... ])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let new_subscriber = match parse_subscriber(form.0) {
        Ok(subscriber) => subscriber,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    match insert_subscriber(&pool, &new_subscriber).await {

```

```

    Ok(_) => HttpResponse::Ok().finish(),
    Err(_) => HttpResponse::InternalServerError().finish(),
}
}

```



The refactoring gives us a clearer separation of concerns:

- `parse_subscriber` takes care of the conversion from our *wire format* (the url-decoded data collected from a HTML form) to our *domain model* (`NewSubscriber`);
- `subscribe` remains in charge of generating the HTTP response to the incoming HTTP request.

The Rust standard library provides a few traits to deal with conversions in its `std::convert` sub-module. That is where `AsRef` comes from!

Is there any trait there that captures what we are trying to do with `parse_subscriber`?

`AsRef` is not a good fit for what we are dealing with here: a *fallible* conversion between two types which **consumes** the input value.

We need to look at `TryFrom`:

```

pub trait TryFrom<T>: Sized {
    /// The type returned in the event of a conversion error.
    type Error;

    /// Performs the conversion.
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

```

Replace `T` with `FormData`, `Self` with `NewSubscriber` and `Self::Error` with `String` - there you have it, the signature of our `parse_subscriber` function!


Let's try it out:

```

//! src/routes/subscriptions.rs
// No need to import the TryFrom trait, it is included
// in Rust's prelude since edition 2021!
// [ ... ]

impl TryFrom<FormData> for NewSubscriber {
    type Error = String;
}

```



```

fn try_from(value: FormData) -> Result<Self, Self::Error> {
    let name = SubscriberName::parse(value.name)?;
    let email = SubscriberEmail::parse(value.email)?;
    Ok(Self { email, name })
}

#[tracing::instrument([ ... ])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let new_subscriber = match form.0.try_into() {
        Ok(form) => form,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}

```

We implemented `TryFrom`, but we are calling `.try_into()`? What is happening there?
 There is another conversion trait in the standard library, called `TryInto`:

```

pub trait TryInto<T> {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}

```

Its signature mirrors the one of `TryFrom` - the conversion just goes in the other direction!
 If you provide a `TryFrom` implementation, your type automatically gets the corresponding `TryInto` implementation, for free.

`try_into` takes `self` as first argument, which allows us to do `form.0.try_into()` instead of going for `NewSubscriber::try_from(form.0)` - matter of taste, if you want.

Generally speaking, what do we gain by implementing `TryFrom`/`TryInto`?

Nothing shiny, no new functionality - we are "just" making our *intent* clearer.

We are spelling out "This is a type conversion!".



Why does it matter? It helps others!

When another developer with some Rust exposure jumps in our codebase they will immediately spot the conversion pattern because we are using a trait that they are already familiar with.

Summary

Validating that the email in the payload of `POST /subscriptions` complies with the expected format is good, but it is not enough.

We now have an email that is *syntactically* valid but we are still uncertain about its *existence*: does anybody actually use that email address? Is it reachable?

We have no idea and there is only one way to find out: sending an actual email.

Confirmation emails (and how to write a HTTP client!) will be the topic of the next chapter.

As always, all the code we wrote in this chapter can be found on [GitHub](#).

Previously

[Using Types For Invariants](#)

Next up

[Writing A REST Client](#)

Buy The Book

This article is a sample from [Zero To Production In Rust](#), a hands-on introduction to backend development in Rust.

You can get a copy of the book at zero2prod.com.

Footnotes

▼ Click to expand!

¹ The `validator` crate follows the HTML specification when it comes to email validation.

You can check its source code if you are curious to see how it's implemented.

Book - Table Of Contents



► Click to expand!



LUCA PALMIERI

