**Write your first scraper in Rust**

[[](https://medium.com/@hmufraggi?source=post_page-----09d9a643a27f--------------------------------)](https://medium.com/@hmufraggi?source=post_page-----09d9a643a27f--------------------------------)

[Mufraggi Hugo](https://medium.com/@hmufraggi?source=post_page-----09d9a643a27f--------------------------------)

·

Follow

5 min read

·

Dec 28, 2023

72

2

Recently, I completed the Rust Programming course at Duke University on [Coursera](https://www.coursera.org/specializations/rust-programming). If you have the time, the specialization is cool and provides valuable advice. It has inspired me to start writing software in Rust. You should create a job scheduling system in Rust.

**Why?**

As I search for my future apartment to buy, I want to save time in tracking the latest announcements. My goal is to streamline my life by creating software that synchronizes data sources. I will use my dataset to write additional articles in the future.

I’ve divided my article into three steps:

1. The scraping part.
2. The creation of the Mongo repository.
3. Assembling everything for the cron.

**Scraping**

Before diving into coding, I need to find a suitable announcement website. I must gain a strong understanding of scraping or protection bypass. Fortunately, I found a website without robust anti-scraping measures. I scraped the website of [le Figaro immobilier](https://immobilier.lefigaro.fr/annonces/immobilier-vente-bien-nice+06100.html), a French media outlet, and focused my search on Nice in the south of France.

**Specifications**

This article focuses solely on scraping the URL <https://immobilier.lefigaro.fr/annonces/immobilier-vente-bien-nice+06100.html.>

To ensure reusability of the code, we delve into the trait system of Rust.

Inside the scraper trait, we define three functions:

1. One for making the call to the URL and transforming the result into HTML.
2. Second, for extracting start and end indices for the “{“ and “}” inside a string.
3. Third, for processing the data and extracting the list of data.

We organize the code in this first step and create the scraper folder and the scraper.rs in src.

src  
├── main.rs  
└── scraper  
 ├── fig\_imo  
 │ ├── domain  
 │ │ ├── list\_announcement.rs  
 │ │ └── mod.rs  
 │ ├── list\_announcement.rs  
 │ └── mod.rs  
 ├── mod.rs  
 └── scraper.rs

**Scraper trait**

After analyzing the data, I decided to split my code into three principal functions, and I will define them in my trait. Moreover, two of the three functions are the same all the time. The definition inside the trait is the perfect place for me.

We need to use the async\_trait package. Currently, the Rust development teams are working to implement async\_trait natively in Rust. You can find more details in this article on Rust's [dev blog](https://blog.rust-lang.org/inside-rust/2023/05/03/stabilizing-async-fn-in-trait.html).

**Trait definition**

I love pattern-matching and use it all the time. I find the usage very easy for managing error handling and dealing with the borrow checker.

#[async\_trait]  
pub trait Scraper<T> {  
 async fn scrape(&self, url: &str) -> Result<Html, ScraperError> {  
 match get(url).await {  
 Ok(response) => {  
 match response.status().is\_success() {  
 true => {  
 let body = response.text().await.unwrap();  
 Ok(Html::parse\_document(&body))  
 }  
 false => {  
 Err(ScraperError(String::from("incorrect web site response")))  
 }  
 }  
 }  
 Err(\_) => {  
 Err(ScraperError(String::from("incorrect web site response")))  
 }  
 }  
 }  
  
 fn extract\_index(&self, text: String) -> Result<(usize, usize), ScraperError> {  
 let start\_index = text.find('{');  
 let end\_index = text.rfind('}');  
 match (start\_index, end\_index) {  
 (Some(start), Some(end)) => {  
 Ok((start, end))  
 }  
 \_ => {  
 Err(ScraperError(String::from("not found { }")))  
 }  
 }  
 }  
  
 fn extract(&self, html: Html) -> Result<Vec<T>, ScraperError>;  
}

**scrape**

In Scrape, we want to do the same logic every time.

match get(url).await {  
 Ok(response) => {  
 match response.status().is\_success() {  
 true => {  
 todo!()  
 }  
 false => {  
 todo!()  
 }  
 }  
 }  
 Err(\_) => {  
 Err(ScraperError(String::from("incorrect web site response")))  
 }  
 }

In the first pattern-matching match get(url).await, we manage the return of the get. Get returns a Result, which has two states: Ok and Err. Err can be many errors. In my catch of the Err, we don’t differentiate between the errors. In the second match response.status().is\_success(), We manage the return of is\_success(), which returns a bool.

true => {  
 let body = response.text().await.unwrap();  
 Ok(Html::parse\_document(&body))  
 }

Inside my true, we get the response body and return Result Ok() with the HTML parsed.

**extract\_index**

ExtExtract\_index is very simple; it takes the HTML content in string format. In the first part, the string contains a JSON with the data of the announcement and the summary in string format.

We chose to extract the index of the first occurrence of { and the last }.

In fact, the code is pretty simple, and we continue to use pattern matching to manage the returns of text.find("{") and text.rfind('}').

**extract**

For extract, we define just the signature of the function, and when we implement the trait in struct, we define the logic.

**Domain**

Before defining extract, we want to define the T return for the fn extract(&self, html: Html) -> Result<Vec<T>, ScraperError>;. We start by creating a new folder src/scraper/fig\_imo/domain, which we use for storing. After inspecting the data returned by the endpoint and with the help of ChatGPT, we define the definition of the struct. We define that:

use serde::Deserialize;  
  
#[derive(Debug, Deserialize)]  
struct FloorSize {  
 #[serde(rename = "@type")]  
 floor\_type: String,  
 value: f32,  
 unitCode: String,  
}  
  
#[derive(Debug, Deserialize)]  
struct Address {  
 #[serde(rename = "@type")]  
 address\_type: String,  
 addressLocality: String,  
 addressRegion: String,  
 postalCode: String,  
}  
  
#[derive(Debug, Deserialize)]  
struct GeoCoordinates {  
 #[serde(rename = "@type")]  
 geo\_type:  
  
 String,  
 addressCountry: String,  
 latitude: f64,  
 longitude: f64,  
 postalCode: String,  
}  
  
#[derive(Debug, Deserialize)]  
pub struct BlockAnnounce {  
 #[serde(rename = "@context")]  
 context: String,  
 #[serde(rename = "@type")]  
 announce\_type: String,  
 name: String,  
 url: String,  
 floorSize: FloorSize,  
 address: Address,  
 geo: GeoCoordinates,  
 image: String,  
}

**list\_announcement**

For this last step, we create list\_announcement.rs in fig\_imo. Inside this file, we want to define the struct for implementing the scraper trait and the extract.

pub struct ListAnnouncementScraper {}  
  
#[async\_trait]  
impl Scraper<BlockAnnounce> for ListAnnouncementScraper {  
 fn extract(&self, html: Html) -> Result<Vec<BlockAnnounce>, ScraperError> {  
 let mut list\_announcement: Vec<BlockAnnounce> = vec![];  
 todo!()  
 }  
}

We define the struct, and we imply the Scraper trait in the struct. At the moment to define impl Scraper<BlockAnnounce>, we define T as BlockAnnounce;

After observing the HTML structure of the website, I decided to target two HTML classes:

* .cartouche-liste.cartouche-liste--polpo
* .cartouche-liste

We will create two selectors and after merging the result [html.select](<http://html.select>)().

let target\_selector = Selector::parse(".cartouche-liste.cartouche-liste--polpo").unwrap();  
let target\_selector2 = Selector::parse(".cartouche-liste").unwrap();  
for bloc in html.select(&target\_selector).chain(html.select(&target\_selector2)) {  
 todo!()  
}  
Ok(list\_announcement)

Now we just have to define the logic. For extracting the data block by block and pushing the result inside the list\_announcement.

let \_bloc\_announce\_text = bloc.text().collect::<String>();  
match self.extract\_index(\_bloc\_announce\_text.clone()) {  
 Ok((start, end)) => {  
 let json\_part = &\_bloc\_announce\_text[start..=end];  
 let block\_announce: BlockAnnounce = serde\_json::from\_str(json\_part).unwrap();  
 list\_announcement.push(block\_announce)  
 }  
 Err(\_) => {  
 continue;  
 }  
}

And we have finished.

For the test in the [main.rs](<http://main.rs>), we just have to initialize the ListAnnouncementScraper, call scrape, and extract.

let s = ListAnnouncementScraper {};  
 let html = s.scrape("<https://immobilier.lefigaro.fr/annonces/immobilier-vente-bien-nice+06100.html>").await.unwrap();  
 let res = s.extract(html).unwrap();  
 for x in &res {  
 println!("{:?}", x)  
 };

**Conclusion**

In the next article, we will see how to define and test a Mongo repository. I hope it interests you, and I look forward to seeing you in the next articles.