We all love how Rust permits us to write fast, secure software. The stabilization of async/.await additionally coincides nicely with another event. I’m excited about the launch of the latest entries in the mainline Pokémon games, Pokémon Sword and Shield, and I’ll make do through fetching data from the PokéAPI for now.

But before diving into the coding part, let’s cover some basic concepts of asynchronous programming and how it may be a bit unique in Rust . But why write asynchronous code?

Asynchronous code permits us to run more than one tasks concurrently on the same OS thread. In a traditional threaded application if you desired to download two different webpages at the same time, you would spread the work throughout two distinct threads, like this:



### This works great for many applications-- after all, threads have been designed to do simply this: run multiple different tasks at once. However, they additionally come with some limitations. There's a lot of overhead involved in the process of switching between distinct threads and sharing data between threads. Even a thread which simply sits and does nothing makes use of valuable system resources. These are the expenses that asynchronous code is designed to eliminate.

[**What is Async/.awaite?**](https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html#why-async)

We can rewrite the function above using Rust's async/.await notation,.

So, what is async await? **Async-await is a way to write functions that can "pause", return control to the runtime, and then pick up from where they left off. Typically those pauses are to wait for I/O, but there can be any number of uses.**

In Rust, values that are ‘awaitable’ are known as ‘futures’. The major aspect you may observe is that futures experience "lazy": they don't do anything till you await them.

use async\_std::task;

// ^ we need this for task spawning

async fn negate\_async(n: i32) -> i32 {

println!("Negating {}", n);

task::sleep(std::time::Duration::from\_secs(5)).await;

println!("Finished sleeping for {}!", n);

n \* -1

}

async fn f() -> i32 {

let neg = negate\_async(1);

// ... nothing happens yet

let neg\_task = task::spawn(negate\_async(2));

// ^ this task /is/ started

task::sleep(std::time::Duration::from\_secs(1)).await;

// we sleep for effect.

neg.await + neg\_task.await

// ^ this starts the first task `neg`

// and waits for both tasks to finish

}

So in the above little code, here’s what’s going on.

* The first line imports async\_std::task we need an external library to run futures as the standard library does not come with an *executor*. This module is similar to [std::thread](https://doc.rust-lang.org/std/thread), except it uses asynchronous tasks in place of threads.
* The async function negate\_async takes as input a signed integer, sleeps for 5 seconds, and returns the negated version of that integer.
* The async function f is more interesting:
  + The first line (let neg ...) creates a Future of the negate\_async function and assigns it to the neg variable. but, it does *not* start execution.
  + The next line of code (let neg\_task ...) uses the task::spawn function to start executing the Future returned by negate\_async. Like with neg, the Future returned by negate\_async is assigned to the neg\_task variable.
  + Next: we sleep for a second. This is so that it will be obvious from the output when a task starts running.
  + Finally, we await both futures, add them together, and return them. By awaiting neg, we start executing the Future and run it to completion. Since neg\_task has already been started, we just wait for it to finish.

So what’s the result of this, then?

Negating 2

# <- there's a 1 second pause here

Negating 1

Finished sleeping for 2!

Finished sleeping for 1!

As we can see, the second future, neg\_task, started executing as soon as it was called—thanks to task::spawn—while neg did not start executing until it was awaited.

### Zero-cost futures

The other distinction between Rust futures and futures in JS and C# is that they are based totally on a "poll" model, which makes them zero cost. In other languages, invoking an async function at once creates a future and schedules it for execution: awaiting the future isn't essential for it to execute. But this implies some overhead for every future that is created.

## A async example!

Alright, let’s get practical.

**Step 1: creating the application**

Run this command in your cmd to create a new project called “async-basics” and open it in VScode

cargo new async-basics

cd async-basics

code .

### Step 2: Dependencies

We’re going to be using [async-std](https://crates.io/crates/async-std) for spawning tasks, and [surf](https://crates.io/crates/surf) to fetch data from the API. Let’s add them to the Cargo.toml file. Your whole file should look something like this:

[package]

name = "async-basics"

version = "0.1.0"

authors = ["Your Name <your.email@provider.tld>"]

edition = "2018"

[dependencies]

async-std = "1"

surf = "1"

### Step 3: Fetch data

Okay, final step. Let’s modify the main.rs file.. Here’s what we want to use:

use async\_std::task;

use surf;

// fetch data from a url and return the results as a string.

// if an error occurs, return the error.

async fn fetch(url: &str) -> Result<String, surf::Exception> {

surf::get(url).recv\_string().await

}

// execute the fetch function and print the results

async fn execute() {

match fetch("https://pokeapi.co/api/v2/move/surf").await {

Ok(s) => println!("Fetched results: {:#?}", s),

Err(e) => println!("Got an error: {:?}", e),

};

}

fn main() {

task::block\_on(execute());

// ^ start the future and wait for it to finish

}

That’s all the code you need. Let’s walk through it!

**use statements:** Just importing the crates we declared in the Cargo.toml file:

* **Surf** is a friendly HTTP client built for casual Rustaceans and veterans alike. It's completely modular, and built directly for async/await
* **async**-**std** is a foundation of portable Rust software, a set of minimal and battle-tested shared abstractions for the broader Rust ecosystem. It offers std types, like Future and Stream , library-defined operations on language primitives, standard macros, I/O and multithreading, among many other things.

**fetch:** This is simply a thin wrapper around the surf::get function which returns either the payload as a String or an Exception if something went wrong.

**Execute:** This function calls fetch with the endpoint for the move Surf, waits for the end result to return, and then fits on the result. If the entirety went well: print the output. Else: print the error.

**Main:** main simply kicks off execute and waits for it to finish. task::block\_on is a synchronous counterpart to task::spawn that starts an asynchronous operation, but blocks until it has finished. Because the main function can’t itself be async (at least not at the time of writing), we can’t use .await in it, but we can block on asynchronous operations.

### Summary

We believe that having async-await on stable Rust is going to be a key enabler for a lot of new and exciting developments in Rust. If you've tried Async I/O in Rust in the past and had problems -- particularly if you tried the combinator-based futures of the past -- you'll find [async-await integrates much better with Rust's borrowing system](http://aturon.github.io/tech/2018/04/24/async-borrowing/). Moreover, there are now a number of great runtimes and other libraries available in the ecosystem to work with. So get out there and build stuff!