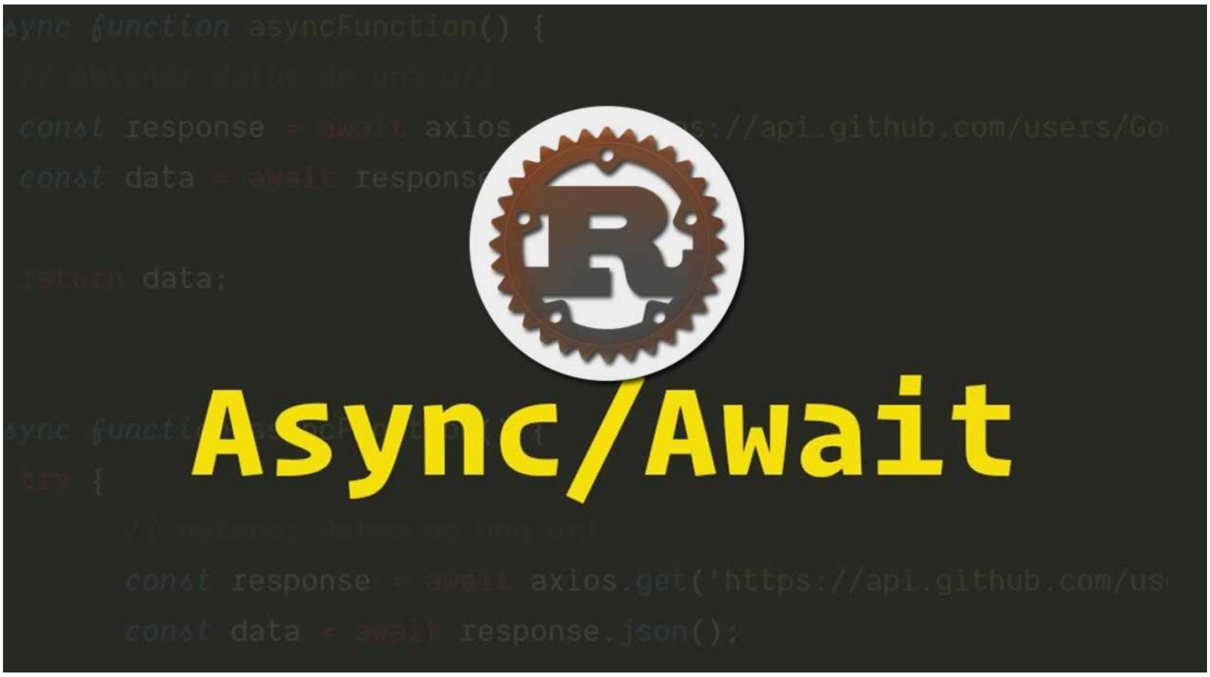
An Article on Async Rust



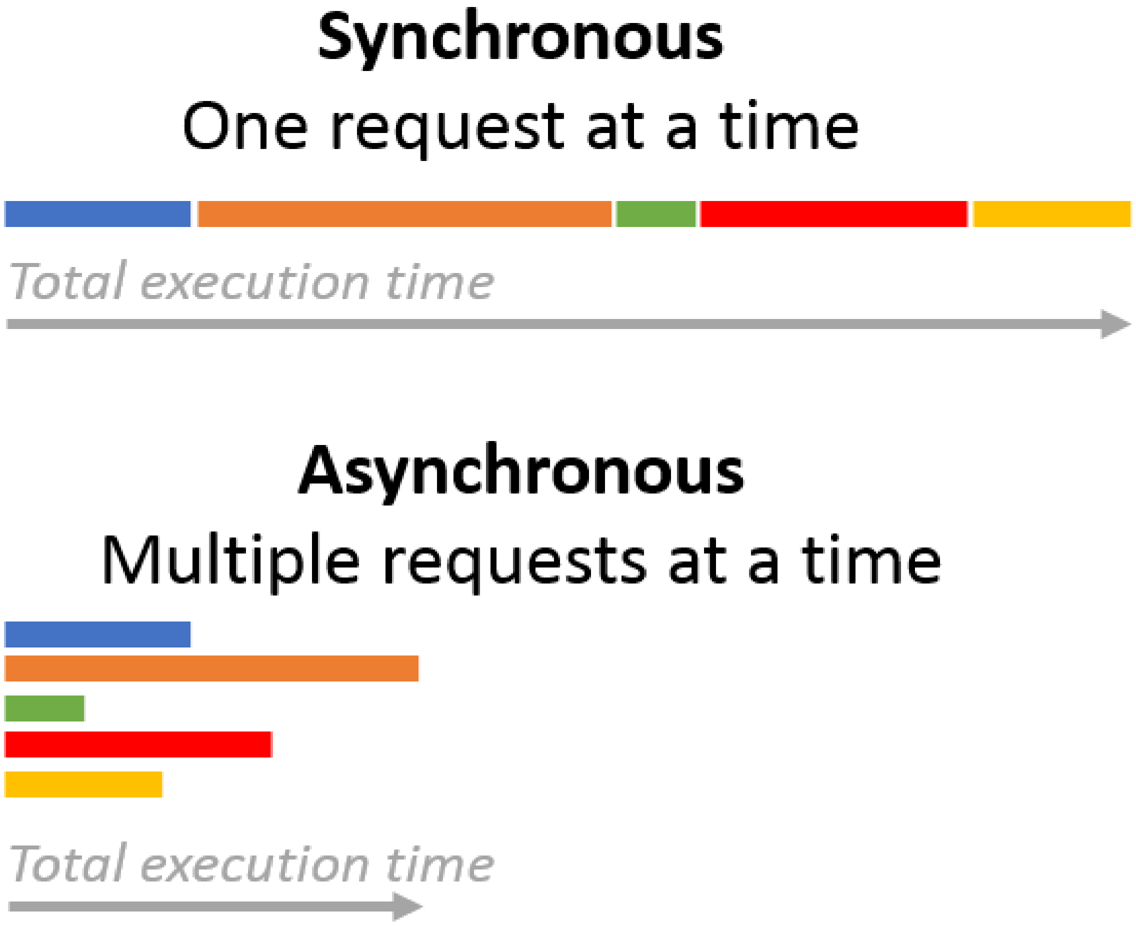
What is Async?

Async is short for “asynchronous”. Async is a mean of running code concurrently. Also, it is meant to be multiple operations running in a time on a same OS thread.

In other words, Asynchronous programming is a parallel programming in which a unit work of an application run separately from the main application, and notifies to the calling thread after completion.

Why Async?

Async programming allows us to run multiple of these IO bound computations at a time on a single thread. Or we can say that, an asynchronous code let us to run several tasks simultaneously on the same OS thread.



Async Rust

Asynchronous coding in Rust Programming Language may be little bit different as it done in other programming languages like C# or JavaScript. In Async Rust, there is fearless concurrency while running multiple operations.

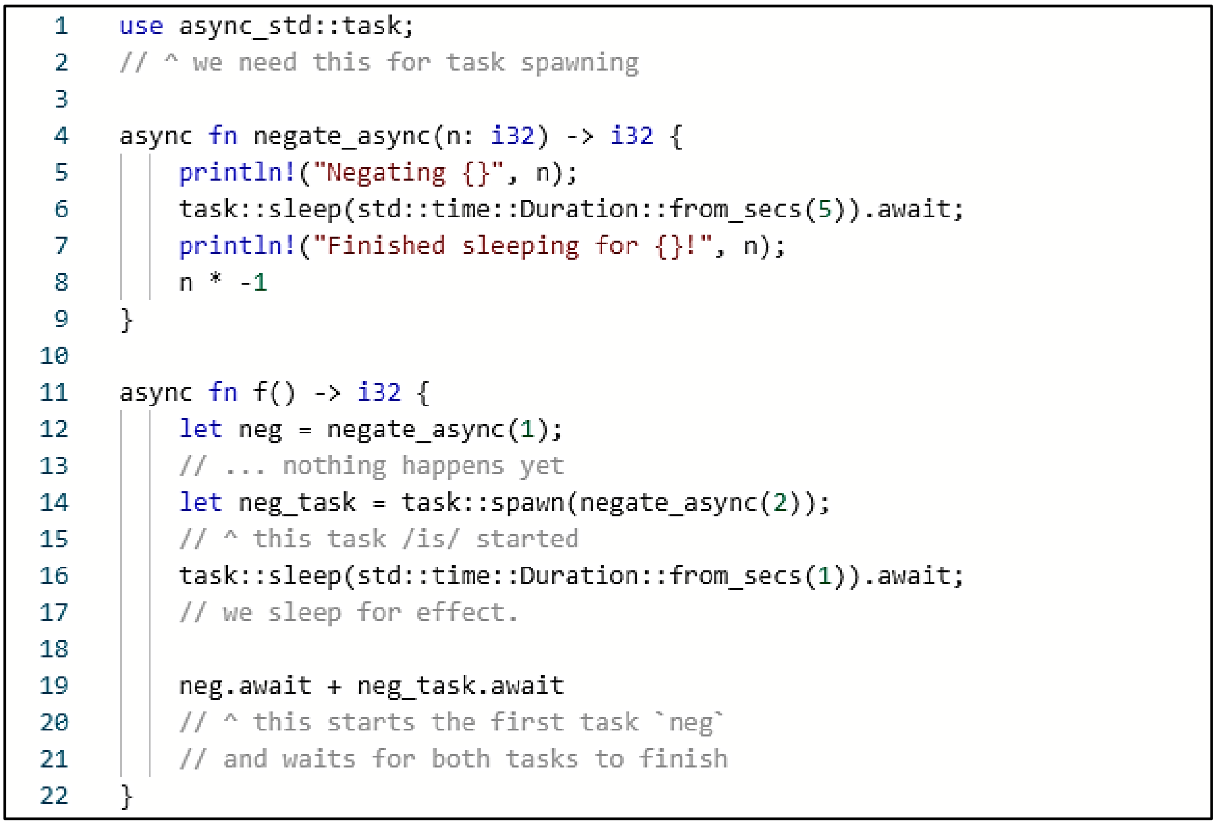
In Rust an Asynchronous function starts by `.await` or by launching a task using an executor.

But the standard library doesn’t come with an executer so, we need an external library to run futures. The executor takes care of executing the futures, polling them and returning the results after completion.

What is meant by Future?

In Rust, `async fn` creates an asynchronous function which returns a Future. To execute the body of the function, the returned Future must be run to completion. They aim to break code into small, composable actions that can be executed by a part of our system.

To understand it, these are few things that must be understood, so consider the following example.



```rust
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
}
```

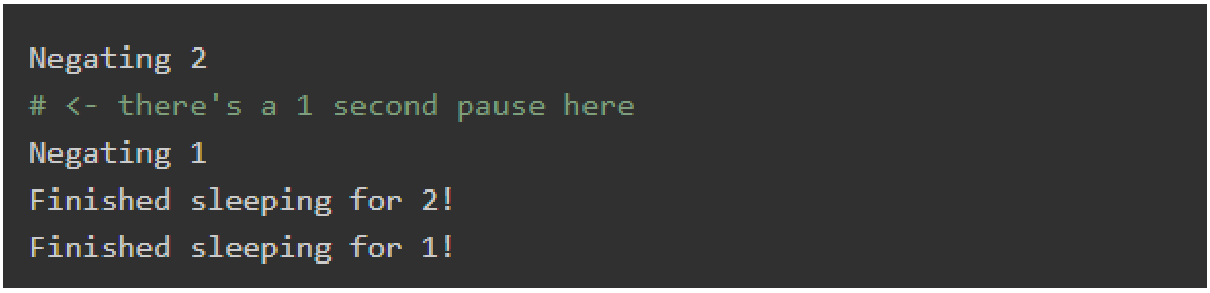
In above Code:

The first line, `async_std::task`.

The async function `negate_async` takes a signed integer as an input, delays for five seconds and returns the number multiplied by -1.

In async function `f`, the first line makes a future of `negate_async` function and assign this to the variable `neg`. The things to be noted that it does not start executing yet. Then in the next line, the `task::spawn` function to start the execution of `Future` returned by negate_async and assign to the neg_task variable. Then to make it obvious before start running the task, it sleeps for a second. In the last, await both the futures and return them after add them together. From neg.await it start executing the future and by neg_task.await it just wait for it to finish because it has already been started.

The output will be:

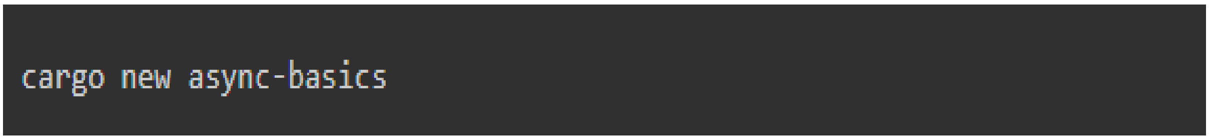


```
Negating 2
# <- there's a 1 second pause here
Negating 1
Finished sleeping for 2!
Finished sleeping for 1!
```

As it shows that, the second future, neg_task, started executing as it is called because of task::spawn, while neg didn't start executing until it was awaited.

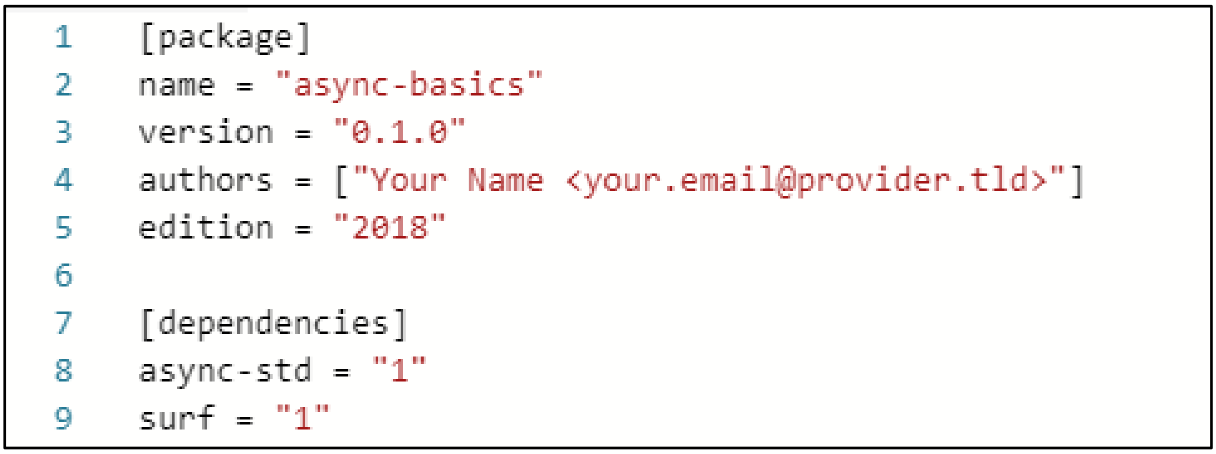
To create a simple application that fetches some data and prints it to the console, the steps are as under:

STEP:1 Create an application by simple running this command.



```
cargo new async-basics
```

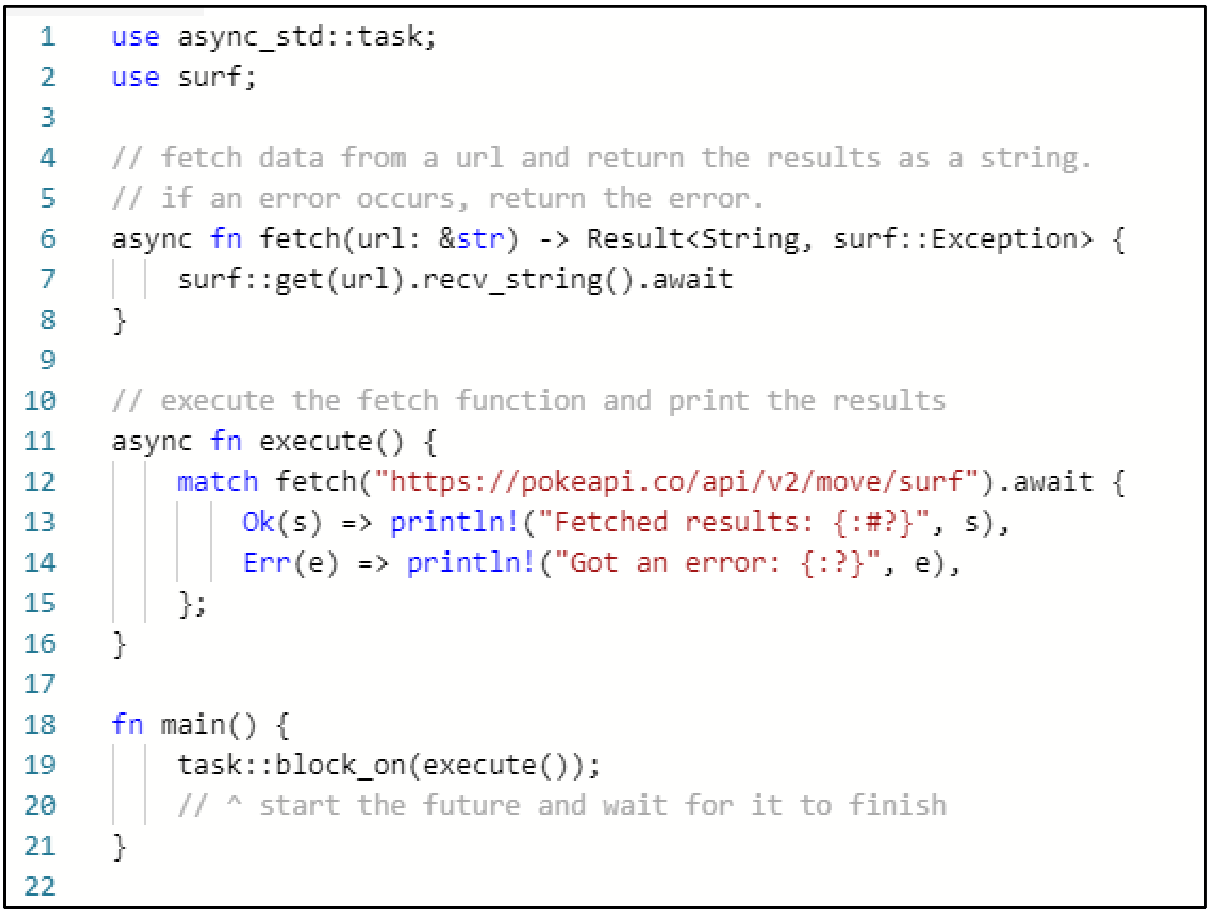
STEP:2 Write async_std for spawning task and surf to fetch data from the API in cargo.toml file. It should look like this.



```toml
[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 Finally, modify the `main.rs` file by using the following code.



```rust
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
}

```

Conclusion

Through Asynchronous Rust programming we can run multiple IO bounds computations concurrently and fearlessly. We need external library to run the futures because the standard library does not have an executer. In rust programming language to create an asynchronous function, `async fn` is used.