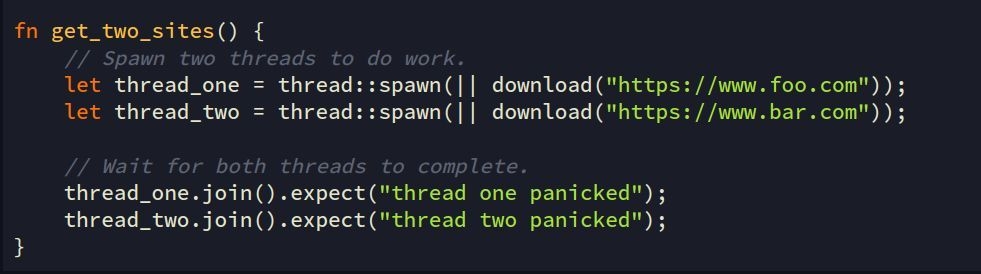
**INTRODUCTION TO ASYNC PROGRAMMING**

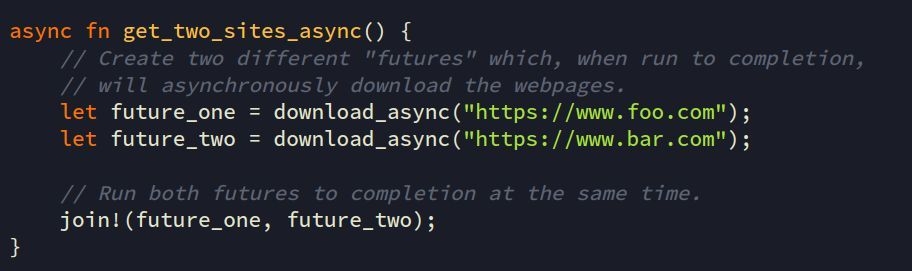
Async (or asynchronous) programming is a proficiency to run multiple operations in our applications inside a single thread. Whatever the nature of your application a web server, a database or an operating system, using async programing you can get the most out of the underlying hardware.

In Rust, when we talk about async, we’re talking about running code concurrently, or having multiple operations running on a single thread. Multi-threading is a related, but a different concept. Multithreading is ideal for when you’ve got computationally intensive tasks that can be spread across multiple cores

In a multi-threaded application, if you wanted to download two different webpages at the same time, you would spread the work across two different threads. For example:

Since threads are designed to run multiple tasks at a same time, they still have some limitations which are mentioned below.

* There's a lot of difficulties in switching between different threads and sharing data between them.
* Even a thread which just sits and does nothing also use valuable system resources.

These are the overheads that asynchronous Rust is designed to reduce. We can rewrite the above function using Rust's async notation. Which will allow us to run multiple tasks at a time without generating multiple threads. For example:

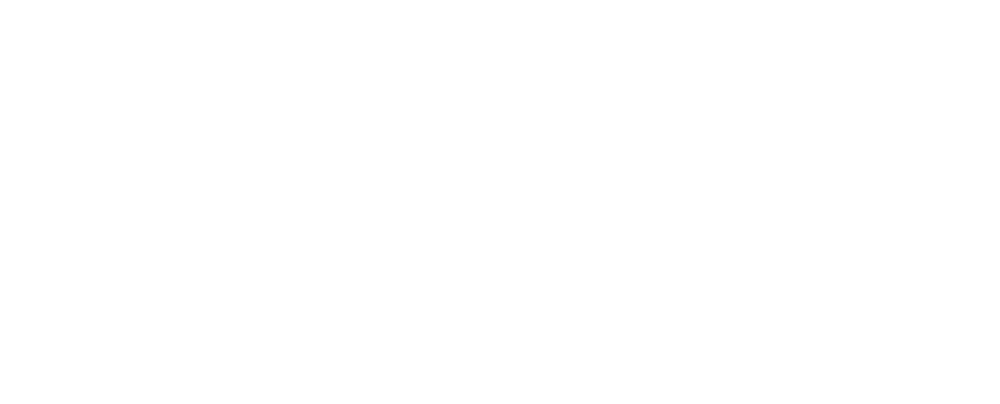
So asynchronous programming lets us run multiple calculations at the same time along a single threads. These operations can run simultaneously because some of them are in waiting for a response, and are in idle state. So the computer can work on something that is not waiting. Whenever the result of an async calculation is required, we must “.await” it. In Rust, those values which are awaitable are known as “futures”.

Overall, asynchronous applications have the potential to be a good deal faster and employ fewer resources than a corresponding threaded implementation. However, asynchronous functions requires special support from the language or libraries. The increased complication of the asynchronous programming model isn't always perfect. It's essential to remember that multi-threaded applications can be quite effective, you just have to consider whether your application can serve better by using the traditional threaded model or the innovative asynchronous model.

As it was mentioned above, you will need an external library to do asynchronous programming in Rust. In Rust, however, you need a dedicated executor. The executor is what manages the execution of the futures, which includes polling them and returning the results when they’re done. The standard library does not contain an executor, so we need an external crate for this. There are a few ones to pick from, but the two most prominent ones are “async-study” and “tokio”.



**async-std**



Asynchronous Rust environment has experienced a lot of evolutions over time, so it can be tough to identify what tools to use, what libraries to invest in, or what documentation to read However, the Future trait inside the standard library and the async/.await language feature has recently been stabilized. The entire ecosystem is in the center of migrating to the newly-stabilized API. However, the ecosystem is still experiencing rapid development as the asynchronous Rust experience is still unpolished. In short, Rust is on its way for having some of the most performant and ergonomic support for asynchronous programming.

“Async/.await” is Rust's built-in tool for writing asynchronous functions that look similar to synchronous code. This async transforms the block of code into a state machine on which the “future” trait is implemented. When we call a “Blocking Function” in a synchronous code, it will block the whole threads whereas in asynchronous code when a “Blocking Function” is called, it will return the control of the thread, allowing other “futures” to run.

The values returned by async function are known as “futures”. In order to complete a task, futures must be run on an executor. The syntax to create an asynchronous function is illustrated in following image.



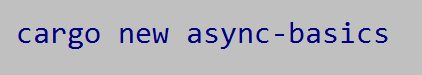
“Block\_on” is one of the async executor which blocks the working thread until the provided future has run to completion. While other executors provide more complex behavior, just like scheduling multiple futures onto the same thread.



Here, we have another async tool “.await”, which works similar to “block\_on” but instead of blocking the whole thread it just wait for a specific future, and allows the other futures/tasks to perform.

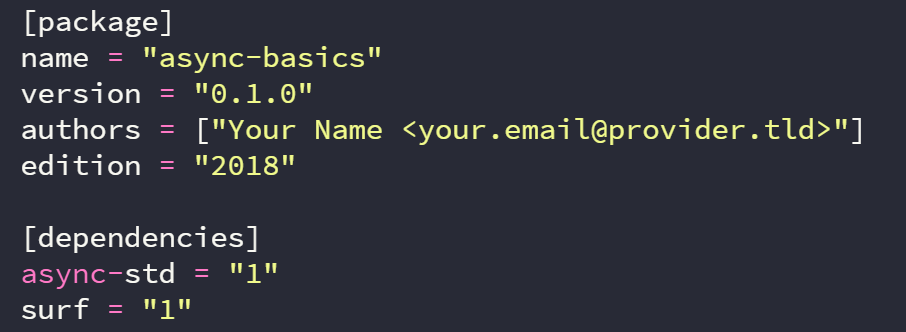
**EXAMPLE**

**STEP 1:**

****First create a new project using **cargo** **new**.

**STEP 2:**

Now we will add two dependencies/crates to our code, one is **async-std** and another is **surf**. Here **async-std** is used for spawning our tasks and **surf** is used for fetching data from the API. Both of the dependencies must be added in **cargo.toml** file as shown below



**STEP 3:**

The final step is to modify our **main.rs** file, it should be as simple as the following image.

Here are some keywords which I would like to explain:

**use** statements

Nothing new, we just use this keyword to import crates that we declared in **cargo.toml** file that are **async-std** and **surf** for this case.

**fetch**

It’s like a wrapper around the **surf::get** which returns something either a payload as a **String** or an **Exception** if anything went wrong.

**execute**

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

**main**

**main** simply run **execute** and waits until it is finished. **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**, we can’t use **.await** in it, however we can block on asynchronous operations.

MADE BY : MIRZA ZIYAD AHMED BAIG ;

IOT-047142 ;