perplexity

give small script that represent that kind of syntax, not necessarily to be tokio future

Certainly! Below is a small Rust script that demonstrates the implementation of a `Future` using the `std::future::Future` trait without relying on an async runtime like Tokio. Instead, it uses a simple struct to represent a future that resolves after a delay:

Rust Script Example

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::{Duration, Instant};
use std::thread;

struct Delay {
    start: Instant,
    duration: Duration,
}

impl Delay {
    fn new(duration: Duration) -> Self {
        Delay {
            start: Instant::now(),
            duration,
        }
    }
}

// Implementing the Future trait for Delay
impl Future for Delay {
    type Output = Result<String, String>;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let this = self.get_mut();

        // Check if the delay duration has passed
        if this.start.elapsed() >= this.duration {
            Poll::Ready(Ok("Delay completed".to_string()))
        } else {
            // Register the current task to be woken up when it's ready
            cx.waker().wake_by_ref();
            Poll::Pending
        }
    }
}
```

```
fn main() {
    // Create a delay future for 2 seconds
    let delay_future = Delay::new(Duration::new(2, 0));

    // Simulate polling the future in a loop
    let mut delay_future = Box::pin(delay_future);

    loop {
        match delay_future.as_mut().poll(&mut Context::from_waker(futures::task::noop_wak
            Poll::Ready(result) => {
                match result {
                    Ok(message) => println!("{}", message),
                    Err(error) => println!("Error: {}", error),
                }
                break; // Exit the loop once ready
            },
            Poll::Pending => {
                println!("Still waiting...");
                thread::sleep(Duration::from_millis(500)); // Sleep to avoid busy waiting
            },
        }
    }
}
```

Explanation

1. **Delay Struct**:
   - Represents a future that completes after a specified duration.
2. **Future Implementation**:
   - The `poll` method checks if the specified duration has elapsed.
   - If it has, it returns `Poll::Ready` with a success message.
   - If not, it returns `Poll::Pending`, indicating that the future is still in progress.
3. **Main Function**:
   - Creates an instance of `Delay` for 2 seconds.
   - Uses a loop to poll the future until it is ready.
   - Prints messages indicating whether it is still waiting or if the delay has completed.

Running the Script

To run this script, ensure you have Rust installed and set up. Save the code in a file named `main.rs`, and then run:

```
cargo run
```

This will simulate an asynchronous operation using a simple custom future implementation without any external async runtime.