Make Your Own Stream Operators

Building Custom Stream Combinators in Rust

Willem Vanhulle

EuroRust 2025

30 minutes + 10 minutes Q&A

About me

- Willem Vanhulle, Software Engineer from Ghent, Belgium
- Specializing in safe, high-performance systems programming
- Founder of SysGhent.be - systems programming community in Ghent
- Author of `clone-stream` crate for cloneable streams
- Languages: Rust, Haskell, Julia + formal verification (Agda, Coq, Lean)

Find me at `github.com/wvhulle` *or* `willemvanhulle.tech`

What we'll cover

Part 1: Foundations

- What are `Streams`?
- Basic consumption patterns
- Existing combinators

Part 2: Building Custom

- The wrapper pattern
- Implementing `Stream` trait

Part 3: Real Example

- Clone-stream library walkthrough
- Memory management gotchas

Part 4: Next Steps

- Advanced patterns
- Resources for learning more

Goal: Build your own stream operators with confidence!

Part 1: Foundations

What are `Streams`?

Think of them as "async `Iterators`" - values that arrive over time:

```rust
// Iterator: all values available immediately
let numbers: Vec<i32> = vec![1, 2, 3, 4, 5];
for n in numbers { /* process sync */ }

// Stream: values arrive asynchronously
let stream: impl Stream<Item = i32> = /* ... */;
while let Some(n) = stream.next().await { /* process async */ }
```

Perfect for network data, user events, sensor readings, etc.

When to use `Streams`?

Perfect for:

- Multiple values arriving over time
- Async data sources (network, files, timers)
- Processing pipelines with transformations
- Event handling (user input, sensors)

Key benefits:

- Lazy evaluation - only process what you need
- Composable - chain operations like `Iterator`
- Async-friendly - doesn’t block other tasks

Understanding async: `Poll`

Before diving into `Streams`, you need to understand `Poll`:

```
enum Poll<T> {
    Ready(T),
    Pending,
}
```

- **Ready** - “Here’s your data!”
- **Pending** - “Check back later, I’m still working on it”

This is how all async operations communicate their state

Simplified `Stream`

A stream is “a future that may be polled more than once”:

```
trait Stream {
    type Item;

    fn poll_next(
        &mut self
    ) -> Poll<Option<Self::Item>>;
}
```

- `Poll::Ready(Some(item))` ⟶ yielded a value
- `Poll::Ready(None)` ⟶ stream is exhausted
- `Poll::Pending` ⟶ not ready, try again later

Synchronous Iterator:

T	Action	Result
1	`(1..=10)`	
2	`next()`	
3		`Some(1)`
4	`next()`	
5	…	
6	`next()`	
7		`None`
8	`next()`	
9		`Some(2)`

→

Asynchronous Stream:

T	Action	Await	Result
1	`St::new()`		
2	`next()`		
3		`await`	
4			`Some(1)`
5	`next()`		
6	…		
7		`await`	
8			`Some(2)`
9	`next()`		
10			`None`

Key difference: Timing

From the comparison we just saw:

`Iterator`

- Synchronous - values ready immediately
- `next()` returns instantly
- Predictable timing

`Stream`

- Asynchronous - values arrive over time
- `next().await` might suspend
- Unpredictable timing - that's the key!

This timing unpredictability is what makes `Streams` perfect for real-world async data

Processing

Process items one by one:

```rust
async fn consume_stream<S>(mut stream: S)
where S: Stream<Item = i32>
{
    while let Some(item) = stream.next().await {
        println!("Processing: {}", item);
    }
}
```

An **imperative** way to handle streams.

Collection

```rust
use futures::stream::{self, StreamExt};
```

Collect all items at once:

```rust
let numbers = stream::iter(vec![1, 2, 3, 4, 5]);
let result: Vec<i32> = numbers.collect().await;
// result: [1, 2, 3, 4, 5]

let sum = stream::iter(1..=10).fold(0, |acc, x| async move
{ acc + x }).await;
// sum: 55
```

Reactivity

```rust
use futures::stream::{self, StreamExt};
```

Familiar operators:

```rust
let result: Vec<_> = stream::iter(1..=10)
    .filter(|&x| async move { x % 2 == 0 })  // Keep evens
    .map(|x| x * 2)                          // Double them
    .take(3)                                 // Take first 3
    .collect()
    .await;

// result: [4, 8, 12]
```

Tokio broadcast `Stream`

Needs helper library `tokio_stream`.

```
use tokio::sync::broadcast;
use tokio_stream::wrappers::BroadcastStream;
use futures::stream::StreamExt;
```

Consume receiving end with a stream wrapper:

```
let (tx, rx) = broadcast::channel(16);
let mut stream = BroadcastStream::new(rx);
```

Producer

Simulating a real producer:

```
tokio::spawn(async move {
    for i in 0..5 {
        tx.send(format!("Message {}", i)).unwrap();
        tokio::time::sleep(Duration::from_millis(100)).await;
    }
});
```

Could be on a different task or machine.

Consumer

Process messages as they arrive

```
stream
    .map(Result::ok)
    .filter_map(future::ready)
    .for_each(|msg| async move {
        println!("Processing: {}", msg);
    })
    .await;
```

Use `Result::ok` and `futures::ready` to ignore broadcast errors.

Part 2: Building custom operators

Step 1: Create a wrapper struct around an existing stream

The wrapper pattern - most custom operators follow this structure:

```rust
struct Double<S> {
    stream: S,  // Wrap the inner stream
}

impl<S> Double<S> {
    fn new(stream: S) -> Self {
        Self { stream }
    }
}
```

Step 2: Implement `Stream` for your wrapper

```
impl<S> Stream for Double<S>
where S: Stream<Item = i32>
{
    type Item = i32;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>> {
        // Implementation goes here...
    }
}
```

The `Pin` challenge

This naive approach doesn't work:

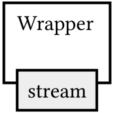
```
fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
    -> Poll<Option<Self::Item>>
{
    let this = self.get_mut(); ❌ Violates Pin contract
    // this.stream is not pinned but needs to be!
}
```

Why it fails: `get_mut()` requires `Self: Unpin`, but our wrapper might not be `Unpin` if the inner stream isn't

Pin projection explained

The problem: We have `Pin<&mut Wrapper>` but need `Pin<&mut InnerStream>`

Memory Layout:



Pin Projection:

`Pin<&mut Wrapper>`

↓

`Pin<&mut stream>`

Pin projection safely converts pinned references without breaking “never move” guarantee

Simple solution: Box the inner stream

Avoid Pin projection complexity by making everything `Unpin`:

```
struct Double<S> {
    stream: Box<S>,  // Box<T> is always Unpin
}

impl<S> Double<S> {
    fn new(stream: S) -> Self {
        Self { stream: Box::new(stream) }
    }
}
```

Why this works: `Box<T>` is always `Unpin`, so `self.get_mut()` is safe

Trade-off: Extra heap allocation vs Pin projection complexity

Define a **blanket implementation** for `Double`:

```
trait StreamExt: Stream {
    fn double(self) -> Double<Self>
    where
        Self: Sized + Stream<Item = i32>,
    { Double::new(self) }
}

impl<S: Stream> StreamExt for S {}
```

Now you can easily double the values in a stream:

```
let doubled = stream::iter(1..=5).double();
```

Part 3: Real Example

Real problem: `Streams` aren't `Clone`

You can't copy a stream like other Rust values:

```
let numbers = stream::iter(vec![1, 2, 3, 4, 5]);
let copy = numbers.clone(); // Error!
```

But sometimes you need multiple consumers:

- Process data in parallel
- Split stream for different tasks
- Cache results for replay

My `clone-stream` crate solves this:

```rust
use clone_stream::ForkStream;

let numbers = stream::iter(vec![1, 2, 3, 4, 5]).fork();
let copy = numbers.clone(); // Works!
```

Now you can:

```rust
let stream1 = numbers.clone();
let stream2 = numbers.clone();
```

Both get the same items: [1, 2, 3, 4, 5]

How clones work

Each clone has its own reading position in the shared buffer:

Buffer	1	2	3	4	5
Clone A	👆				
Clone B			👆		

- Clone A will read `1` next
- Clone B will read `3` next
- Both share the same buffer data
- Each tracks their own position independently

Waker coordination

Clone-stream creates a “meta-waker” that wakes all waiting clones:

```rust
fn waker(&self, extra_waker: &Waker) -> Waker {
    let wakers = self.clones
        .iter()
        .filter_map(|(_id, state)| state.waker())
        .collect::<Vec<_>>();

    MultiWaker::new(wakers)
}
```

Only waiting clones contribute their wakers

How waking works

The coordination process:

1. **Clone waits** - Clone calls `.next().await`, returns `Pending`
2. **Waker stored** - Clone's waker gets stored in its state
3. **Base stream polled** - Meta-waker given to base stream
4. **New data arrives** - Base stream wakes the meta-waker
5. **All wake up** - Meta-waker wakes all waiting clones

Efficient: no unnecessary wake-ups for clones that aren't waiting

```rust
let original = stream::iter(vec![1, 2, 3, 4, 5]).fork();

let evens = original.clone()
    .filter(|&x| async move { x % 2 == 0 });

let doubled = original.clone()
    .map(|x| x * 2);

// Both process the same source data independently
let (even_results, doubled_results) = tokio::join!(
    evens.collect::<Vec<_>>(),
    doubled.collect::<Vec<_>>()
);
```

Clone-stream usage

Both clones get all items:

```rust
use clone_stream::ForkStream;

let original = stream::iter(vec!['a', 'b', 'c']).fork();
let mut adam = original.clone();
let mut bob = original.clone();

// Both receive 'a'
assert_eq!(adam.next().await, Some('a'));
assert_eq!(bob.next().await, Some('a'));
```

Each clone maintains its own position

Smart buffering: Setup

Bob polls first, but no data is available yet:

```rust
let (sender, rx) = tokio::sync::mpsc::unbounded_channel();
let stream = tokio_stream::wrappers::UnboundedReceiverStream::new(rx);

let mut adam = stream.fork();
let mut bob = adam.clone();

// Bob starts waiting for data (no data sent yet)
let bob_task = tokio::spawn(async move {
    bob.next().await // Returns Pending, Bob gets suspended
});
```

Bob is now in a “waiting” state with his waker stored

Smart buffering: Data arrives

Adam polls and data arrives - this wakes Bob too:

```rust
// Meanwhile, data gets sent
sender.send('a').unwrap();

// Adam polls and gets the data
let adam_task = tokio::spawn(async move {
    adam.next().await // Gets Some('a') immediately
});

// Bob's waker gets triggered automatically!
let (adam_result, bob_result) = tokio::join!(adam_task, bob_task);

assert_eq!(adam_result.unwrap(), Some('a'));
assert_eq!(bob_result.unwrap(), Some('a')); // Same data!
```

How it works

The coordination:

1. Bob waits → waker stored
2. Adam polls → data arrives
3. Meta-waker → wakes both
4. Both get same data

Buffering only happens when clones are actively waiting

Late cloning

You can clone even after receiving some items:

```rust
let (sender, rx) = tokio::sync::mpsc::unbounded_channel();
let stream = tokio_stream::wrappers::UnboundedReceiverStream::new(rx);

let mut adam = stream.fork();

sender.send('a').unwrap();
assert_eq!(adam.next().await, Some('a'));

// Clone after adam already read 'a'
let mut bob = adam.clone();

sender.send('b').unwrap();
assert_eq!(bob.next().await, Some('b')); // Bob gets the next item
```

Memory management warning

⚠️ Suspended clones can cause memory buildup:

```
let original = some_big_stream().fork();

let mut fast = original.clone();
let mut very_slow = original.clone();

// very_slow gets suspended waiting for data
let slow_task = tokio::spawn(async move {
    very_slow.next().await // Suspended!
});
```

Now very_slow is in a suspended state

Memory buildup problem

Fast reader can't clean up items. `fast` reader processes 1000s of items:

```
for _ in 0..1000 {
    fast.next().await;
}
```

All 1000 items still buffered! `very_slow` hasn't read them yet

Solution: Drop unused clones

```
drop(slow_task); // Frees buffered memory
```

Part 4: Conclusion

Next steps

Exercises:

- `timeout(duration)` - cancel slow streams
- `batch(n)` - group items into chunks
- `rate_limit(per_second)` - throttle stream speed

Other topics:

- `stream::unfold` for simple stream states
- Nested streams with `flatten` combinators
- The complementary `Sink` trait
- Reactive futures crate: `futures-rx`

Summary

- Streams are async iterators that return multiple values at unpredictable times
- Start with existing combinators - `map`, `filter`, `collect`, `fold`
- Build custom operators using the wrapper pattern + `Stream` trait
- Clone-stream enables parallel processing - but watch memory with slow readers

You can now build your own stream operators!

Questions?

📖 Blog series: `wvhulle.github.io/blog/streams/`

📦 Clone-stream: `github.com/wvhulle/clone-stream`

📚 Futures docs: `docs.rs/futures`

Willem Vanhulle • `@wvhulle` • EuroRust 2025