QUICKWIT

A Deep Dive into Tower

`async fn(Req) -> Result<Resp, Error>`

Hello 👋

- Adrien Guillo (@guilload)
- Developing in Rust for ~3 years, mostly contributing to Quickwit
- Slides at guilload.com/fosdem-2024

What is Tower?

- crate for building **modular** networking clients and servers
- widely used within the Rust ecosystem (`axum`, `warp`, `tonic`, ...)
- based on the `Service` trait

Why do we need Tower?

In an imaginary dynamic language, we could write this...

```
def get_user(request):
    logging.info(f"started processing request {request.method} {request.path}")
    user = Users.get(request.username)
    response = Response.ok(user)
    logging.info("finished processing request")
    response
```

However, it would be better to write that...

```
def with_logging(handler)(request):
    logging.info(f"started processing request {request.method} {request.path}")
    response = handler(request)
    logging.info("finished processing request")
    response

def get_user(request):
    user = Users.get(request.username)
    Response.ok(user)

get_user_with_logging = with_logging(get_user)
```

*We want to write **generic** and **reusable** functions that are easy to **compose**.*

Decorator pattern

- A function that wraps a function
- Applies additional behavior before or after the inner function
- In the context of clients and servers, often called **middlewares**

```
def with_behavior(handler)(request):
    # Insert behavior before processing the request...
    response = handler(request)
    # and/or after processing the request.
    response
```


Request
Response
Middleware 1
Middleware 2
Middleware n
Handler

Fairly easy to implement with dynamic languages:

- duck typing gives us great **flexibility**
- decorated functions must **agree implicitly** on their input and output types

How can we compose functions in a type-safe and flexible manner in Rust?

The `tower::Service` trait

Allows implementing components in a **protocol-agnostic and composable** way.

```
pub trait Service<Request> {
    type Response;
    type Error;
    type Future: Future<Output = Result<Self::Response, Self::Error>>;

    fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>>;

    fn call(&mut self, req: Request) -> Self::Future;
}
```

*“Just” a **generic async function***

```rust
/// Processes a request and returns a response asynchronously.
async fn call(&mut self, request: Request) -> Result<Response, Error>;
```

*"Just" a **generic async function**... with a twist!*

```
fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>>;
```

- `poll_ready` must be called before `call`
- provides a way to propagate backpressure

A `poll_ready` implementation for a service without external dependencies:

```
fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>> {
    Poll::Ready(Ok(()))
}
```

A `poll_ready` implementation for a service with a database dependency:

```
fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>> {
    if self.conn_opt.is_none() {
        self.conn_opt = Some(futures::ready!(self.pool.poll_acquire(cx))?);
    }
    Poll::Ready(Ok(()))
}
```

A `poll_ready` implementation for a middleware:

```
fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>> {
    self.inner.poll_ready(cx)
}
```

It sounds simple on paper, so why does it get complex?

- Lots of generics
- Rust idiosyncrasies (lifetimes, `Send` + `Sync` marker traits, ...)
- Exposure to advanced concepts such as future polling or pinning

> “The best way out is always through.”

Robert Frost

Let's implement a `Hello` service!

```rust
#[derive(Debug)]
struct HelloRequest(String);

#[derive(Debug)]
struct HelloResponse(String);

async fn hello(request: HelloRequest) -> HelloResponse {
    let message = format!("Hello, {}!", request.0);
    HelloResponse(message)
}
```

We define a `Hello` *struct and start implementing* `Service` *for it:*

```rust
#[derive(Debug)]
struct Hello;

impl Service<HelloRequest> for Hello {
    type Response = HelloResponse;
    type Error = Infallible;
    type Future = ?;

    ...
}
```

Choosing a `Future` type

1. **Boxed future**

For instance, `futures::future::BoxFuture`:

```rust
type BoxFuture<'a, T> = Pin<Box<dyn Future<Output = T> + Send + 'a>>;
```

Why choosing a boxed `Future`?

Pros:

- easy
- readable

Cons:

- allocation
- dynamic dispatch

Good choice for applications, less for libraries

We opt for `BoxFuture` *:*

```
impl Service<HelloRequest> for Hello {
    type Response = HelloResponse;
    type Error = Infallible;
    type Future = BoxFuture<'static, Result<Self::Response, Self::Error>>;

    ...
}
```

We start implementing `poll_ready`:

```rust
impl Service<HelloRequest> for Hello {
    type Response = HelloResponse;
    type Error = Infallible;
    type Future = BoxFuture<'static, Result<Self::Response, Self::Error>>;

    fn poll_ready(&mut self, _cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>> {
        Poll::Ready(Ok(()))
    }
}
```

Finally, we implement `call`*:*

```rust
impl Service<HelloRequest> for Hello {
    type Response = HelloResponse;
    type Error = Infallible;
    type Future = BoxFuture<'static, Result<Self::Response, Self::Error>>;

    fn call(&mut self, request: HelloRequest) -> Self::Future {
        let future = async move {
            let message = format!("Hello, {}!", request.0);
            HelloResponse(message)
        };
        Box::pin(future)
    }
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use tower::ServiceExt;

    #[tokio::test]
    async fn test_hello_service() {
        let response = Hello
            .ready()
            .await
            .unwrap()
            .call(HelloRequest("Alice".to_string()))
            .await
            .unwrap();
        assert_eq!(response.0, "Hello, Alice!");
    }
}
```

Choosing a `Future` type

1. Boxed future
2. **Reuse named future from third-party crate (`futures`, `tower`)**

For instance, `futures::future::Ready`.

```rust
use futures::future::{ready, Ready};

impl Service<HelloRequest> for Hello {
    type Response = HelloResponse;
    type Error = Infallible;
    type Future = Ready<Result<Self::Response, Self::Error>>;

    fn call(&mut self, request: HelloRequest) -> Self::Future {
        let message = format!("Hello, {}!", request.0);
        let response = HelloResponse(message);
        ready(response)
    }
}
```

Let's implement a `Logging` service!

`Logging` decorates an inner service `S`:

```
#[derive(Debug)]
pub struct Logging<S> {
    inner: S
}
```

We start implementing `Service` *for* `Logging`*:*

```
impl<S, R> Service<R> for Logging<S>
where
    S: Service<R>, // The inner service must be a `Service`.
{
    ...
}
```

Then, we implement `poll_ready` *:*

```
impl<S, R> Service<R> for Logging<S>
where
    S: Service<R>,
{
    type Response = S::Response;
    type Error = S::Error;
    type Future = BoxFuture<'static, Result<Self::Response, Self::Error>>;

    fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>> {
        self.inner.poll_ready(cx)
    }
}
```

Finally, we implement `call`:

```rust
impl<S, R> Service<R> for Logging<S>
where
    S: Service<R>,
{
    type Response = S::Response;
    type Error = S::Error;
    type Future = BoxFuture<'static, Result<Self::Response, Self::Error>>;

    fn call(&mut self, request: R) -> Self::Future {
        let inner_future = self.inner.call(request);
        let outer_future = async move {
            tracing::info!("started processing request");
            let response = inner_future.await;
            tracing::info!("finished processing request");
            response
        };
        Box::pin(outer_future)
    }
}
```

It should work...

```
error: future cannot be sent between threads safely
  --> src/logging.rs:59:9
   |
59 |         Box::pin(future)
   |         ^^^^^^^^^^^^^^^^ future created by async block is not `Send`
   |
   = help: within `{async block@src/logging.rs:53:22: 58:10}`, the trait `std::marker::Send` is not implemented for `<S as tower::Se
rvice<R>>::Future`
note: captured value is not `Send`
  --> src/logging.rs:55:28
   |
55 |             let response = inner_future.await;
   |                            ^^^^^^^^^^^^ has type `<S as tower::Service<R>>::Future` which is not `Send`
   = note: required for the cast from `Pin<Box<{async block@src/logging.rs:53:22: 58:10}>>` to `Pin<Box<(dyn futures::Future<Output
= Result<<S as tower::Service<R>>::Response, <S as tower::Service<R>>::Error>> + std::marker::Send + 'static)>>`
help: consider further restricting the associated type
   |
51 |     fn call(&mut self, request: R) -> Self::Future where <S as tower::Service<R>>::Future: std::marker::Send {
   |                                                    +++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

`BoxFuture` *is* `Send + 'static` *so* `S::Future` *must be too.*

```
impl<S, R> Service<R> for Logging<S>
where
    S: Service<R>,
    S::Future: Send + 'static, // We added the constraints `Send + 'static` here.
{
    ...
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use tower::ServiceExt;

    #[tokio::test]
    async fn test_logging_service() {
        let mut service = Logging {
            inner: Hello,
        };
        let response = service
            .ready()
            .await
            .unwrap()
            .call(HelloRequest("Alice".to_string()))
            .await
            .unwrap();
        assert_eq!(response.0, "Hello, Alice!");
    }
}
```

Choosing a `Future` type

1. Boxed future
2. Reuse named future from third-party crate (`futures`, `tower`)
3. **Roll our own `Future`**

Rolling our own `Future`

```rust
use pin_project::pin_project;

#[pin_project]
pub struct LoggingFuture<F> {
    #[pin]
    inner: F,
}
```

Rolling our own `Future`

```rust
impl<S, R> Service<R> for Logging<S>
where
    S: Service<R>,
    S::Future: Send + 'static,
{
    type Response = S::Response;
    type Error = S::Error;
    type Future = LoggingFuture<S::Future>;

    fn call(&mut self, request: R) -> Self::Future {
        tracing::info!("started processing request");

        LoggingFuture {
            inner: self.inner.call(request),
        }
    }
}
```

Rolling our own `Future`

```
impl<F> Future for LoggingFuture<F>
where
    F: Future,
{
    type Output = F::Output;

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

        if polled.is_ready() {
            tracing::info!("finished processing request");
        }
        polled
    }
}
```

Let's implement a `Timeout` service!

`Timeout` *decorates an inner service* `S`*:*

```rust
#[derive(Debug)]
pub struct Timeout<S> {
    inner: S,
    timeout: Duration,
}
```

What `Error` type should we return?

```
impl<S, R> Service<R> for Timeout<S>
where
    S: Service<R>,
{
    type Response = S::Response;
    type Error = ?
}
```

Choosing an `Error` type

We must signal the timeout or propagate the error type from the inner service.

```rust
pub enum TimeoutError<E> {
    Timeout,
    Inner(E),
}
```

Choosing an `Error` type

- Hard to compose in practice
- Boxed errors are usually favored

For instance, `tower::BoxError`:

```rust
pub type BoxError = Box<dyn Error + Send + Sync>;
```

We start implementing `poll_ready`:

```rust
impl<S, R> Service<R> for Timeout<S>
where
    S: Service<R>,
    S::Error: Into<BoxError>,
{
    type Response = S::Response;
    type Error = BoxError;
    type Future = TimeoutFuture<S::Future>;

    fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>> {
        match self.inner.poll_ready(cx) {
            Poll::Pending => Poll::Pending,
            Poll::Ready(result) => Poll::Ready(result.map_err(Into::into)),
        }
    }
}
```

Then, we implement `call`:

```rust
impl<S, R> Service<R> for Timeout<S>
where
    S: Service<R>,
    S::Error: Into<BoxError>,
{
    type Response = S::Response;
    type Error = BoxError;
    type Future = TimeoutFuture<S::Future>;

    fn call(&mut self, request: Request) -> Self::Future {
        let inner_future = self.inner.call(request);
        let sleep_future = tokio::time::sleep(self.timeout);

        TimeoutFuture::new(inner_future, sleep_future)
    }
}
```

Finally, we implement `TimeoutFuture` *:*

```rust
#[pin_project]
pub struct TimeoutFuture<F> {
    #[pin]
    inner: F,
    #[pin]
    sleep: Sleep,
}
```

```rust
impl<F, T, E> Future for TimeoutFuture<F>
where
    F: Future<Output = Result<T, E>>,
    E: Into<crate::BoxError>,
{
    type Output = Result<T, crate::BoxError>;

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

        // First, try polling the inner future.
        match this.inner.poll(cx) {
            Poll::Ready(result) => return Poll::Ready(result.map_err(Into::into)),
            Poll::Pending => {}
        }

        // Then, check the sleep future.
        match this.sleep.poll(cx) {
            Poll::Pending => Poll::Pending,
            Poll::Ready(_) => Poll::Ready(Err(Elapsed(()).into())),
        }
    }
}
```

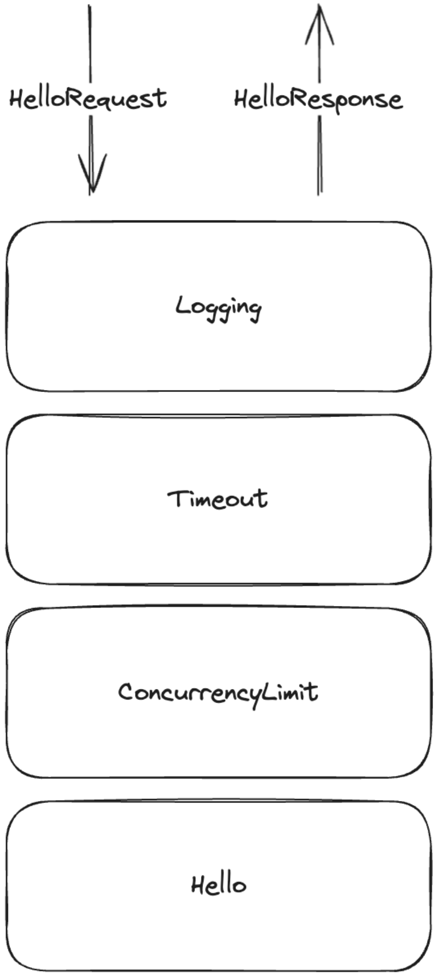
Stacking services

Let's stack some built-in services from the `tower` crate on top of our `Hello` service.

```rust
use std::time::Duration;
use tower::limit::ConcurrencyLimit;
use tower::timeout::Timeout;

let service = Hello;
let service = ConcurrencyLimit::new(service, 5);
let service = Timeout::new(service, Duration::from_secs(5));
let mut service = Logging::new(service);
```

The order with which you wrap your services matters ⚠️


HelloRequest
HelloResponse
Logging
Timeout
ConcurrencyLimit
Hello

Your roadmap to mastering Tower

Learn about:

- `tower::Layer`
- `tower::ServiceBuilder`

Your roadmap to mastering Tower

Read some literature:

- "Inventing the Service trait", blog post by David Pedersen
- `axum` documentation page about middlewares

Your roadmap to mastering Tower

Read some code:

- `tower::limit::RateLimit`
- `tower::limit::ConcurrencyLimit`
- `tonic::transport::Channel`

Your roadmap to mastering Tower

Watch some videos:

- “Rust live coding - Tower deep dive”, David Pedersen, YouTube
- “The What and How of Futures and async/await in Rust”, Jon Gjengset, YouTube

Questions?

guilload.com/fosdem-2024