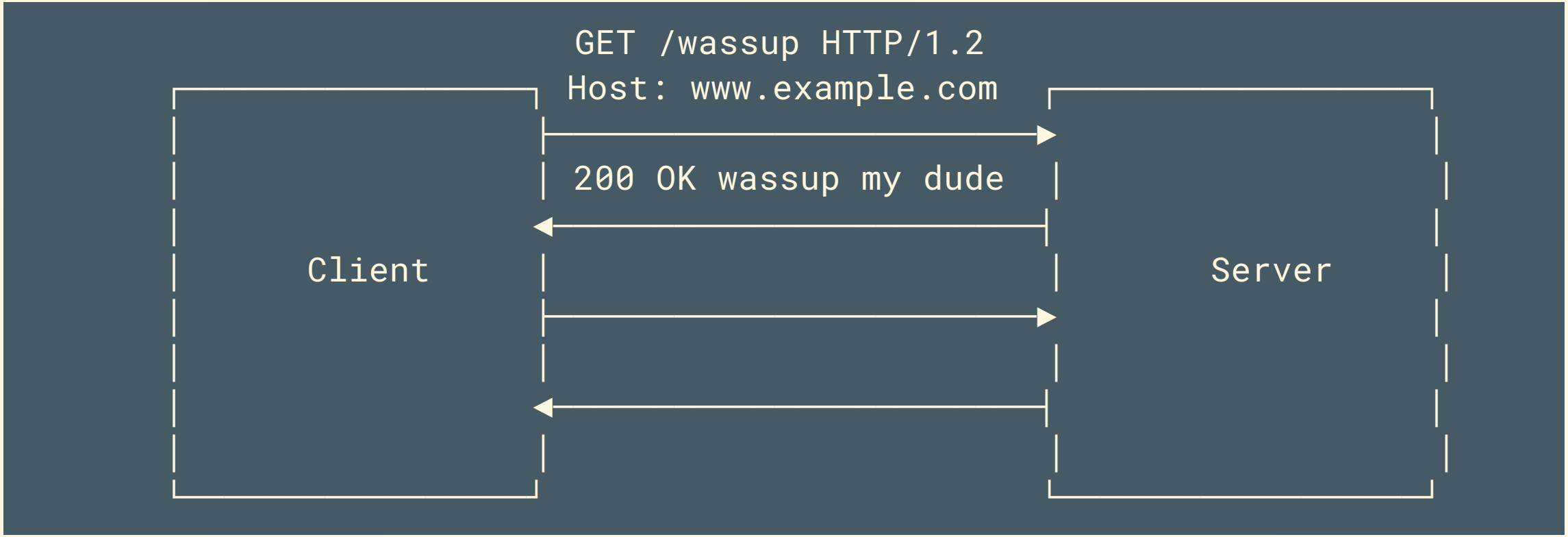
Networking, async, tokio, axum and more



Networking and stuff

- The Rust standard library only provides transport-layer functionality through `TcpStream`, `TcpListener`, `UdpSocket` and friends
- Networking above transport layer is messy and changing
- `reqwest` is a great client-library for sending HTTP-requests
- `axum` is a great server-library for web-servers
- `hyper` is a low-level HTTP-library, also used in Curl
- Networking is *async*hronous by nature

Blocking vs non-blocking

- In an asynchronous environment certain actions take more time than others, e.g. reading from a disk, sending and receiving a packet across half the world
- Those actions may take a *lot* longer time than other common actions (~ 100 billion times longer than an integer addition)
- Threads can either do this action in a *blocking* manner, waiting until the OS reports back that the packet is returned, or a non-blocking manner, which means it can periodically poll for new events (or be notified by the OS of new events), so it can perform other actions in the meantime.
- `async/await` in common languages is just an implementation of that last behavior of non-blocking asynchronous code.
- The main benefit of `async/await` is not performance, but allowing other work to be performed at the same time, such as if a thread *also* renders some UI, a

Async in Rust vs async how you know it

- Async in other languages such as JS and C# is straightforward
- `async Task<int> MyFunc()` can be called directly, can be `await`-ed and can use await internally directly.
  Such as `var x = await MyFunc();`
- There is a hidden cost with this:
  - Async state machines
  - Task-level storage (variables, task/promise progress, etc)
  - Executor, someone's gotta push/pull it forward

Rust gotta go fast

- Hidden costs is Rust's worst enemy. Goes directly against zero-cost abstractions, which is why Rust removed green threads just prior to 1.0 release
- `async` / `await` is built into the language, but with `.await`-postfix syntax for awaiting a value
- `async fn a() -> Stuff` or `fn b() -> impl Future<Output = Stuff>`
- `let x = my_func().await?.do_more().await?;`, providing better reading comprehension
- Abstraction is a friend, so abstract away async engine

Futures, promises, tasks...

```rust
pub trait Future {
    type Output;

    // Required method
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

- `Context` has `Waker` which `wake`s your task somehow
- `Future`s are not started when spawned, unlike JS's `Promise` and C#'s `Task`
- `let x = do_stuff();` where `x` is a `Future` does "nothing"
- Ignore `Pin<T>` for now, it is an implementation detail but it

`async fn` desugared

- Writing converting `fn fun() -> Thing` to `async fn fun() -> Thing` converts the function so it doesn't actually return `Thing` anymore, but `impl Future<Output = Thing> + Sized + 'static`
- The future-type cannot be explicitly named because it *may* capture stack-local values, so we can only write `impl Future<...>`, similarly to closure's `impl Fn() -> Thing`
- This is similar to how other languages does it, except Rust also tracks liveness, so it adds `'static` unless you really want a way around it.

Executors and runtimes

- Is your system bound by a single core? A single (OS) thread? What about embedded?
- Should `Future`s be run concurrently, parallelized or not? Should they be completed on the same thread as they started on?
- Runtimes provide three things (from `tokio`'s documentation):
  - An I/O event loop, called the driver, which drives I/O resources and dispatches I/O events to tasks that depend on them.
  - A scheduler to execute tasks that use these I/O resources.
  - A timer for scheduling work to run after a set period of time.

Popular runtimes

- Tokio
  - Oldest, most-used, provides multithreaded executor and is most likely the one you want in most cases
- Async-std
  - Essentially an async runtime ecosystem to mirror the standard library in an async way
- Smol
  - So basically, it's smol.
  - Small and fast runtime

All are easy to use: Tokio

```rust
#[tokio::main]
async fn main() {
    // This is running on a core thread.

    let blocking_task = tokio::task::spawn_blocking(|| {
        // This is running on a blocking thread.
        // Blocking here is ok.
    });

    // We can wait for the blocking task like this:
    // If the blocking task panics, the unwrap below will propagate the
    // panic.
    blocking_task.await.unwrap();
}
```

Async std

```
async fn say_hello() {
    println!("Hello, world!");
}

#[async_std::main]
async fn main() {
    say_hello().await;
}
```

Smol

```
use smol::prelude::*;

async fn say_hello() {
    println!("Hello, world!");
}

fn main() -> smol::io::Result<()> {
    smol::block_on(async {
        say_hello().await
        Ok(())
    })
}
```

Axum, tower, tokio

- Tokio tries to create an ecosystem and creates Tower
- Tower is a layered HTTP library for services
  - Want to create a server? Implement `Service`
  - Want to create a middleware? Implement `Layer`
- Axum is a webserver-library created on top of Tower
- Uses some type-tricks for ergonomic and safe (tries to be crash-free) DevEx

Axum 101

```rust
#[tokio::main]
async fn main() {
    let app = Router::new()
        .route("/", get(root)) // `GET /` goes to `root`
        .route("/users", post(create_user)) // `POST /users` goes to `create_user`
        .with_state(app_state) // Can be database connection pool, etc.

    let listener = tokio::net::TcpListener::bind("0.0.0.0:3000").await.unwrap();
    axum::serve(listener, app).await.unwrap();
}
async fn root() -> &'static str { "Hello, World!" }
async fn create_user(
    // Parse the request body as JSON into a `CreateUser` type
    Json(payload): Json<CreateUser>,
) -> (StatusCode, Json<User>) {
    // TODO: Application logic
    // this will be converted into a JSON response with a status code of `201 Created`
    (StatusCode::CREATED, Json(user))
}
```

Debugging Axum

Unfortunately, fancy types and ergonomics leads to *worse* compiler error messages, so the Axum team has added `axum::debug_handler`. Just add the macro onto your failing function and it may clarify the error message:

```
#[tokio::main]
async fn main() {
    let app = Router::new()
        .route("/users", post(create_user))
        .with_state(app_state)

    let listener = tokio::net::TcpListener::bind("0.0.0.0:3000").await.unwrap();
    axum::serve(listener, app).await.unwrap();
}
#[axum::debug_handler]
async fn create_user(
    MyFancySerializationLanguage(payload): MyFancySerializationLanguage<CreateUser>,
) -> (StatusCode, Json<User>) {
```