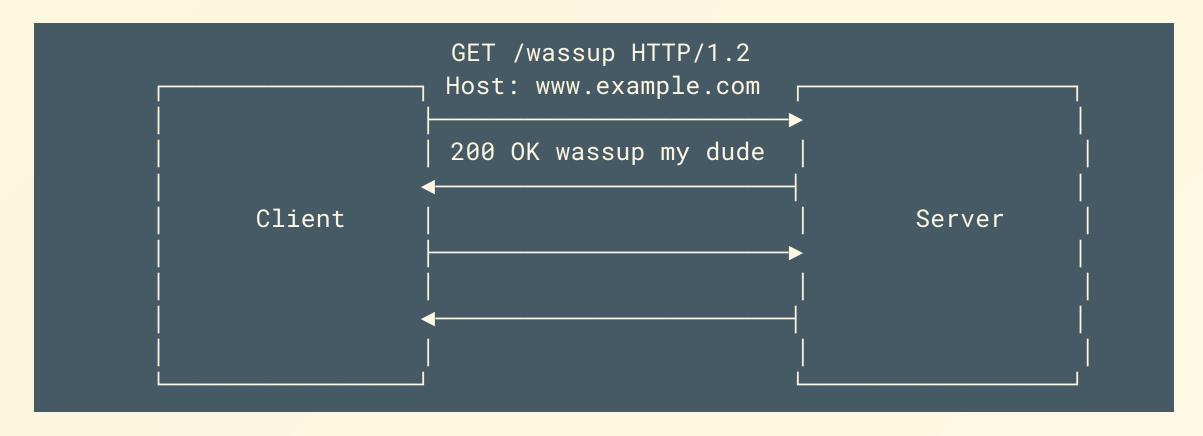
Networking, async, tokio, axum and more



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
- request 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 asynchronous by nature

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...

```
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"

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

```
#[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 crashfree) DevEx

Axum 101

```
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 {
async fn create_user(
    // Parse the request body as JSON into a `CreateUser` type
    Json(payload): Json<CreateUser>,
 -> (StatusCode, Json<User>) {
    // <u>TODO:</u> Application logic
    // this will be converted into a JSON response with a status code of `201 Created`
    (StatusCode::CREATED, Json(user))
```