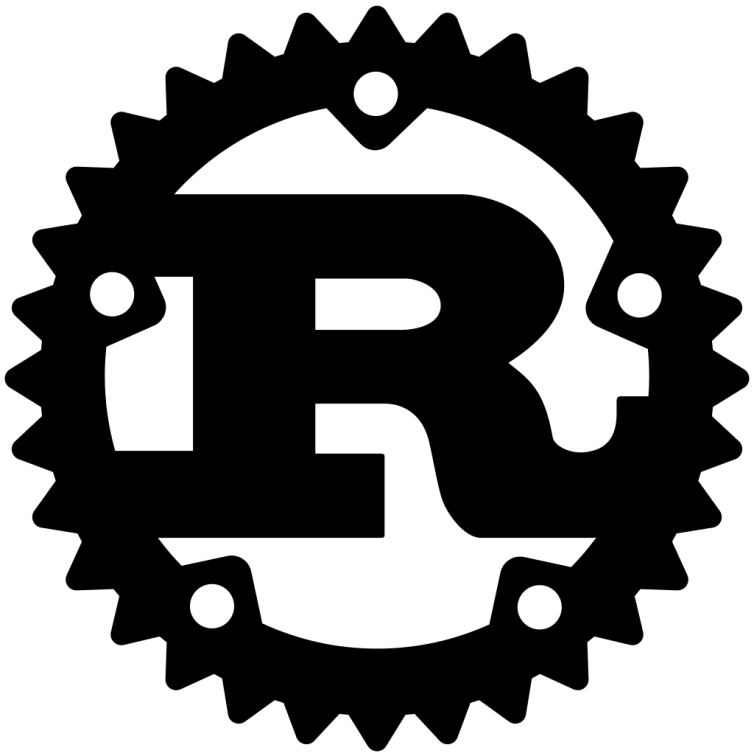

# Programming in Rust

Grzegorz Nosek

# Level 0: the command line

## Objectives

- start your first Rust project
- read command line arguments
- print the first argument to standard output

## Hints

- run cargo init to create your project
- create a file called src/bin/level0.rs with your code (feel free to copy src/main.rs as a starting point)
- create urls.txt containing the URLs we will eventually fetch. For example,

```
https://httpbin.org/200
https://httpbin.org/ip
https://example.com/
https://httpbin.org/200
https://httpbin.org/ip
https://example.com/
https://httpbin.org/200
https://httpbin.org/ip
https://example.com/
https://httpbin.org/200
https://httpbin.org/ip
https://example.com/
https://httpbin.org/200
https://httpbin.org/ip
https://example.com/
```

- run your program with the command:

```
cargo run --bin level0 -- urls.txt
```

- use std::env::args() to get an iterator over the command line arguments
- use the nth() method of trait Iterator to get the first argument
  Note: elements are numbered from zero but the 0th one is the program name
- nth() returns an Option<String> so you will need to .unwrap() it to get to the String inside
- println!("{}", something) formats an object using its Display implementation
- println!("{:?}", something) formats an object using its Debug implementation
- there are solutions for all exercises at https://github.com/gnosek/rust-lab-spoilers but try to do them yourself first :)

## Links

- https://www.rust-lang.org/learn
- https://doc.rust-lang.org/book/
- https://doc.rust-lang.org/rust-by-example/
- https://doc.rust-lang.org/std/env/fn.args.html
- https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.nth
- https://doc.rust-lang.org/std/option/enum.Option.html#method.unwrap

# Level 1: loading files

## Objectives

- read contents of a file line by line
- collect the lines in a vector
- print out the contents of the vector using Debug

## Hints

- use File::open (it takes many different types as its argument, including String)
- you will need to .unwrap() the Result you get from File::open to access the File object inside
- File only implements a basic read operation. There exists a lines() method in the std::io::BufRead trait but File does not implement that trait
- std::io::BufReader is an implementation of BufRead in the standard library
- to use any trait's methods, you need to explicitly use it. You can also use the BufReader struct for convenient access:

```
use std::io::{BufRead, BufReader};
```

- wrap your File object with a BufReader instance:

```
let url_file = BufReader::new(url_file);
```

- note that BufRead::lines() returns an iterator of Results (each individual read may fail, after all); since we're not doing anything with the vector yet, you can either unwrap() each line, or just store the Results in the vector
- Vec<T> is a basic collection type, similar to std::vector<T> in C++. You create an instance by calling Vec::new():

```
let mut urls = Vec::new();
```

- note how you don't need to specify the type, it's inferred from the context of where the vector is later used
- use the push() method to add items to your vector
- print the resulting vector using the {:?} pattern (Vec<T> doesn't implement Display so you can't use {})

## Links

- https://doc.rust-lang.org/std/io/struct.BufReader.html
- https://doc.rust-lang.org/std/io/trait.BufRead.html#method.lines
- https://doc.rust-lang.org/std/vec/struct.Vec.html

# Level 2: basic error handling

## Objectives

- return a Result type from main()
- handle errors by using the Result type and the ? operator instead of unwrap()

## Hints

- our current main() function does not return anything (i.e. it returns the () type)
- it can return Result<(), any-error-type> instead
- all the standard library methods we call so far use std::io::Error as the error type, so it's a good choice for us (for now)
- if you return Ok(()) from main, the program exits successfully, otherwise it will panic with an error message
- the ? operator unwraps any Result, returning the Ok value or propagating the Err value to the caller function
- to convert an Option<T> (like the one you get from nth()) to a Result (that you can handle with ?), use the ok_or method on the Option. It takes an error as an argument so we need to create one
- std::io::Error can be constructed with its new method, which takes two arguments:
  - a std::io::ErrorKind enum to choose the error class
  - a string which will become the error message:

```
let url_path = url_path.ok_or(Error::new(ErrorKind::NotFound,
                                         "File name missing"))?;
```

- we now need to handle the Results returned from the lines() method. There's a nice trick for this, where you can construct a Result<any-collection<T>, E> from an iterator over Result<T, E>. It requires calling the collect() method on the Result type you want
- since the compiler cannot know what collection you want (a Vec, or maybe a HashSet etc.), you need to help the type inference a bit (the element type and the error type can still be deduced):

```
let urls: Result<Vec<_>, _> = url_file.lines().collect();
```

## Links

- https://doc.rust-lang.org/std/io/struct.Error.html
- https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.collect

# Level 3: use an HTTP library

## Objectives

- use the reqwest HTTP library to send requests

## Hints

- to add a library as a dependency, add its name and version to the [dependencies] section in Cargo.toml. We want to use the 0.9 version of reqwest, so the relevant section will look like this:

```
[dependencies]
reqwest = "0.9"
```

- that's it, cargo will download and compile the library the next time you build your project; Rust 2015 edition required that you add extern crate reqwest to your main.rs (or lib.rs if you're building a library) but it's no longer needed in the 2018 edition
- we only need the most basic features of the library, so our HTTP request code will be just:

```
let client = reqwest::Client::new();
let resp = client.get(&url).send()?;
println!("{}", resp.status());
```

- the error type returned by reqwest is not std::io::Error (it's an established convention that every library provides its own error type), so we need to either:
  - map the error to a std::io::Error
  - return "any error type possible" from main()
  - create our own error type that encapsulates either of the two possible error types
- option 3 is what you would do in production code (and we'll take a peek at it later), option 2 is good enough in the short term and option 1 isn't really used
- so, just change the error type that main() returns to Box<dyn std::error::Error> and the ? operator will take care of type conversions
- note that std::io::Error is a concrete type while std::error::Error is a trait and the only way to pass trait objects around is by using a boxed pointer (you can't pass them by value as you don't know their size)

## Links

- https://docs.rs/reqwest/0.9.19/reqwest/struct.Client.html
- https://doc.rust-lang.org/std/error/trait.Error.html

# Level4: collect timings

## Objectives

- create a struct type that expresses the timing of an individual request
  - the time taken to do the request
  - the number of bytes received
- extract our HTTP client to a function
- make the function return an instance of the new struct

## Hints

- std::time::Instant::now() returns the current time using a monotonic clock
- std::time::Duration is a difference between two Instants
- #[derive(Debug)] will give you a Debug implementation for free
- reqwest::Response has a text() method which returns a Result<String, _>
- your get() method will have to return a Result too

## Links

- https://doc.rust-lang.org/std/time/index.html
- https://docs.rs/reqwest/0.9.19/reqwest/struct.Response.html

# Level 5: aggregate the timings

## Objectives

- create a method in your Stats struct to aggregate two timings together
- create a method to calculate the average speed in bytes per second of a request

## Hints

- there are different possible signatures for the aggregate() method, each one expresses a slightly different intent
- fn aggregate(&mut self, other: &Stats) is good enough for us right now
- in the method body, just add the individual fields (use the += operator)
- to calculate the speed in bytes per second, divide the response length by the time taken
- but first, you need to adjust the types. Converting the byte count to f64 just requires an explicit conversion (bytes as f64)
- std::time::Duration has a method that does exactly what we need (.as_secs_f64()) but it's not yet stable, so, we need to do this manually:
  - convert the duration to an integer number of milliseconds first (using .as_millis())
  - convert the milliseconds to f64
  - divide the milliseconds by 1000.0
- since the elapsed time may be shorter than a millisecond, we need to take care to avoid division by zero
- return an Option<f64>, equal to Some(speed) if we can calculate it, None otherwise
- the caller of this method can use unwrap_or(0) or unwrap_or_default() on the resulting Option if zero is an appropriate approximation of infinite speed

## Links

- https://doc.rust-lang.org/std/time/struct.Duration.html

# Level 6: callbacks

## Objectives

- make your get() function take a callback, which takes a Stats object and returns Result<(), box dyn std::error::Error>. get() should now return Result<(), box dyn std::error::Error> as well

## Hints

- all closures have an unnameable type, so your function needs to be generic
- you need to decide the right trait bound. The least restrictive for the callee are the most restrictive for the caller:
  - Fn can't modify its environment but can be called without restrictions
  - FnMut can modify its environment but it's subject to borrowing restrictions (can't mutate the environment outside of the closure while the closure exists)
  - FnOnce can drop its environment so it's callable only once
- all Fn-compatible objects implement FnMut and FnOnce too
- all FnMut-compatible objects implement FnOnce too
- your new get() function should be called like:

```
get(&client, &url, |req_stats| {
    totals.aggregate(&req_stats);
    Ok(())
})?;
```

- the signature of your new get() function will roughly look like this (you will need to change the details):

```
fn takes_callback<F>(i: i32, callback: F) -> i32
where
    F: Fn(i32) -> i32
{
    // ...
}
```

## Links

- https://doc.rust-lang.org/book/ch13-01-closures.html
- https://doc.rust-lang.org/std/ops/trait.Fn.html
- https://doc.rust-lang.org/std/ops/trait.FnMut.html
- https://doc.rust-lang.org/std/ops/trait.FnOnce.html

# Level 7: more callbacks

## Objectives

- calculate the speedup of the (upcoming) parallelism of your code

## Hints

- let's define the speedup as the sum of elapsed time of all requests divided by actual elapsed wall clock time
- create a function called calc_speedup() that takes a closure returning Result<Duration, std::error::Error> and returns Result<f64, std::error::Error>
- nothing new here language-wise

# Level 8: threads

## Objectives

- make your client multi-threaded

## Hints

- std::thread::spawn() takes a closure and executes it in a separate thread
- for every URL, create a separate thread, passing it a closure that calls get()
- we don't need to share the reqwest::Client object across threads, it's easier to create a new one for each thread
- create a vector (call it threads), where you'll keep the join handles returned by std::thread::spawn
- loop over that vector, calling .join() on every element
- your total stats are now accessed across threads so you need to wrap them in Arc<Mutex<>>
- each thread will need a copy of the wrapped stats object (which refers to the same actual Stats instance underneath)
- once your code compiles, it's thread safe

## Links

- https://doc.rust-lang.org/std/thread/index.html

# Level 9: the failure crate

## Objectives

- use the failure crate to encapsulate all possible errors in a single concrete type

## Hints

- why a concrete type? It lets the compiler derive various useful facts about our type that aren't necessarily true for a boxed trait object (e.g. it being Send + Sync + 'static). It will come handy soon.
- add failure (version 0.1) to your dependencies section in Cargo.toml
- create an enum type (call it CacheWarmerError) with three variants:
  - Reqwest, holding a reqwest::Error
  - Io, holding a std::io::Error
  - FileNameMissing, without any data members
- use the failure::Fail trait
- decorate the enum with #[derive(Fail)]
- for each variant, decorate it with #[fail(display = "...")]
- for variants with data members, you can use format strings similar to println!
- you can refer to tuple struct items in the fail macro as _0, _1 etc. Here you'll have at most one, so your attribute might look like this:

```
#[fail(display = "HTTP client error: {}", _0)]
```

- to get the ? operator working seamlessly with your new error type, implement the From<T> trait for your new type twice:

```
impl From<reqwest::Error> for CacheWarmerError {
    fn from(e: reqwest::Error) -> Self {
        CacheWarmerError::Reqwest(e)
    }
}

impl From<std::io::Error> for CacheWarmerError {
    fn from(e: std::io::Error) -> Self {
        CacheWarmerError::Io(e)
    }
}
```

- replace all uses of Box<dyn std::error::Error> with your new type
- use the FileNameMissing variant to handle the case where the program is ran without an argument

## Links

- https://rust-lang-nursery.github.io/failure/
- https://doc.rust-lang.org/std/convert/trait.From.html

# Level 10: combinators

## Objectives

- remove the thread pool and go back to sequential code
- convert your main loop and your get() function to combinators (monadic style)

## Hints

- make all the I/O in your get() function a single expression
- instead of retrieving results with let x = y?, call y.and_then(|x| ...)
- you will need to adjust the error type explicitly, with map_err(|e| e.into()) or map_err(CacheWarmerError::from)
- in the main loop
  - map() the iterator over URLs through a closure that calls get()
  - fold() over the resulting iterator of Result<Stats, CacheWarmerError>
    - the initial value can be e.g. Ok(Stats { elapsed: Duration::default(), content_length: 0 })
    - the closure to combine two results-of-stats can look like this:

```
|a: Result<_, CacheWarmerError>, b| {
    let mut a = a?;
    a.aggregate(&b?);
    Ok(a)
}
```

## Links

- https://fsharpforfunandprofit.com/posts/recipe-part2/
- https://doc.rust-lang.org/std/result/enum.Result.html
- https://doc.rust-lang.org/std/option/enum.Option.html
- https://doc.rust-lang.org/std/iter/trait.Iterator.html
- https://www.youtube.com/watch?v=Des3zZuTbhk

# Level 11: into the future(s)

## Objectives

- make your code asynchronous (though not concurrent yet)
- use reqwest::async::Client instead of reqwest::Client
- use Futures instead of Results and Streams instead of Iterators
- use the tokio runtime to drive the resulting Future to completion

## Hints

- add tokio and futures (version 0.1 of both) to your dependencies
- both the get() function and the stats callback must now return an unnameable type that implements Future<Item=T, Result=CacheWarmerError>
- the return type of get() can be inferred by the compiler using the impl keyword in the return type, but for the closure we need a new type parameter
- the final signature for get() may look e.g. like this:

```
fn get<T, F, FR>(
    client: &reqwest::r#async::Client,
    url: &str,
    stats_callback: F,
) -> impl Future<Item = T, Error = CacheWarmerError>
where
    FR: Future<Item = T, Error = CacheWarmerError>,
    F: FnOnce(Stats) -> FR + 'static,
{
    // ...
}
```

- the async reqwest response has a slightly different API: instead of a text() method it has an into_body() method that converts the response into a Stream of chunks. Use fold() on the resulting stream to sum the lengths of the chunks
- use future::ok to wrap a value in a Future returning that value immediately
- convert your vector of results of URLs to a stream using stream::iter_result, then map that

```
resp.into_body()
    .fold(0, |total, chunk| {
        future::ok::<_, reqwest::Error>(total + chunk.len())
    })
```

  stream through a closure that calls get(). The result is a stream of Future<Item=Stats, Error=CacheWarmerError> objects
- fold() that stream to aggregate the timings. Since each item of the stream is a Future itself, you need to map it through your actual aggregation closure. Here's how it might look like:

```
let totals_fut = requests.fold(Stats::new(),
|mut totals, req_stats| {
    req_stats.map(move |s| {
        totals.aggregate(&s);
        totals
    })
});
```

- the fold() call returns a Future that, when polled to completion, will yield a Result<Stats, CacheWarmerError> object
- finally, start the tokio runtime, passing it the future representing the whole client

```
let mut runtime = tokio::runtime::Runtime::new()
    .expect("Unable to create a runtime");
let totals = runtime.block_on(totals_fut)?;
```

- note: before starting the runtime, not a single line of the future's code has been executed

## Links

- https://tokio.rs/
- https://docs.rs/futures/0.1.28/futures/future/trait.Future.html
- https://docs.rs/futures/0.1.28/futures/stream/trait.Stream.html
- https://docs.rs/reqwest/0.9.18/reqwest/async/struct.Response.html

# Level 12: actual concurrency

## Objectives

- use futures::stream::Buffered to execute futures in parallel

## Hints

- the Stream we just built yields individual Futures that need to be resolved themselves
- this happens sequentially in the .fold() method of the Stream trait
- the .buffered() method on a Stream of Future<Item=T>s converts it to a Stream of Ts, while resolving the futures in parallel
- you can call .fold() on a buffered stream just like on a normal one, but the items are no longer Futures so you can just use the value, without .map()ping the Future that wraps it:

```
.fold(Stats::new(), |mut totals, req_stats| {
    totals.aggregate(&req_stats);
    future::ok::<_, CacheWarmerError>(totals)
});
```

## Links

- https://docs.rs/futures/0.1.28/futures/stream/trait.Stream.html#method.buffered

# Level 13: async/await

## Objectives

- use the nightly toolchain with async/await support for more ergonomic async code

## Hints

- install the nightly toolchain with rustup toolchain add nightly
- create a new cargo project (you'll need different dependencies)
- use the git master branch of all the dependencies
  - use "futures-preview" instead of "futures" and enable the "compat" feature
  - the "runtime" crate (along with "runtime-tokio") gives us "async fn main"

```
[dependencies]
failure = "0.1"
futures-preview = {
    git = "https://github.com/rust-lang-nursery/futures-rs",
    features = ["compat"]
}
reqwest = {
    git = "https://github.com/seanmonstar/reqwest",
    branch = "master"
}
runtime = {
    git = "https://github.com/rustasync/runtime",
    branch = "master"
}
runtime-tokio = {
    git = "https://github.com/rustasync/runtime",
    branch = "master"
}
```

- enable the two nightly-only features needed for async/await support in your .rs file:
- import a few items from the futures crate

```
#![feature(async_await)]
#![feature(async_closure)]
```

  - futures::stream::FuturesOrdered is similar to the futures::stream::Buffered struct in that it resolves futures in parallel, but it doesn't need a Stream to do that, just a collection of Futures
  - futures::stream::StreamExt gives our Streams a .next() method so that it looks more like an iterator
  - Future01CompatExt and Stream01CompatExt are a bridge between the futures 0.1 crate (that reqwest still needs) and the new std::future type (.compat() method)

```
use futures::compat::{Future01CompatExt, Stream01CompatExt};
use futures::stream::{FuturesOrdered, StreamExt};
```

    - you will need to call .compat() on the futures you get from reqwest Client/Response methods

- futures::Future<Item=T, Error=E> now becomes std::future::Future<Output=Result<T, E>>
  - this means that futures no longer correspond to Results directly but instead any type can be computed asynchronously
- all functions returning impl Future<Item=T, Error=E> can now be written as async fn foo(...) -> Result<T, E>
- to resolve a future in async code, use the .await operator. For futures yielding Results, you can use .await?
- to iterate over a stream, you need while let:

```
while let Some(req_stats) = futs.next().await {
    let req_stats = req_stats?;
    totals.aggregate(&req_stats);
}
```

- you can .push() a future into a FuturesOrdered object which then behaves as a Stream of the futures' resolved values
- your main function can be async too, you just need the #[runtime::main] attribute on it
  - right now reqwest::async only works with the tokio runtime, so you need to choose that explicitly:

```
#[runtime::main(runtime_tokio::Tokio)]
async fn main() -> Result<(), CacheWarmerError> {
    // ...
}
```

## Links

- https://rust-lang.github.io/async-book/index.html (we need to .await for more content)
- https://areweasyncyet.rs/
- https://tmandry.gitlab.io/blog/posts/optimizing-await-1/

# Unsafe

## Objectives

- stay as long as you can without using unsafe

## Links

- https://doc.rust-lang.org/nomicon/
- https://rust-unofficial.github.io/too-many-lists/
- http://francismurillo.github.io/2019-07-31-Understanding-Rust-Through-AVL-Trees/

# FFI: embedding Rust code in C++

## Objectives

- build a C++ binary that uses a library written in Rust

## Hints

- clone https://github.com/gnosek/rust-lab-life
- make sure you have cmake 3.12+ and a C++17 compiler
- review the C language interface life-cpp/ffi-import.h and life_rs/ffi_export.rs
- review life-cpp/build-ffi-rust-in-cpp.sh
- run the script
- for larger projects you can generate the C bindings automatically using e.g. rusty-cheddar

## Links

- https://crates.io/crates/rusty-cheddar

# FFI: embedding C++ code in Rust

## Objectives

- build a Rust binary that uses a library written in C++

## Hints

- clone https://github.com/gnosek/rust-lab-life
- make sure you have cmake 3.12+ and a C++17 compiler
- review the C language interface life-cpp/ffi-export.cpp and life_rs/ffi_import.rs
- review life-cpp/build-ffi-cpp-in-rust.sh
- run the script
- for larger projects you can generate the C bindings automatically using e.g. rust-bindgen

## Links

- https://rust-lang.github.io/rust-bindgen/
- https://medium.com/dwelo-r-d/using-c-libraries-in-rust-13961948c72a
- https://medium.com/dwelo-r-d/wrapping-unsafe-c-libraries-in-rust-d75aeb283c65

# DNS: basic synchronous server

## Objectives

- use the provided crate implementing the DNS protocol to write a simple DNS server

## Hints

- create a new project and add the dns crate from https://github.com/gnosek/rust-lab-dns as a dependency
- use the std::net::UdpSocket struct to open a UDP socket
  - you will need the .send_to() and .recv_from() methods to read and write data from the socket
- use the following items from the dns crate:
  - dns::parse::parse_dns_packet(input: &[u8]) -> Result<(&[u8], DnsPacket), DnsParseError>
  - dns::utils::respond(query: DnsPacket, src: &SocketAddr) -> DnsPacket
  - the dns::build::Serialize trait that DnsPacket implements
    - the trait provides one method:
      fn serialize_to(&self, buf: &mut Vec<u8>) -> Result<(), std::io::Error>
- the basic structure of your code should be:
  - create a UDP socket
  - in an endless loop:
    - receive a packet
    - try to parse it
    - if successful, generate a response and send it back

## Links

- https://bodil.lol/parser-combinators/
- https://www.ietf.org/rfc/rfc1035.txt

# DNS: Tokio-based UDP server

## Objectives

- use the provided dns crate to write a Tokio-based UDP server

## Hints

- the tokio UDP echo example is a good place to start:
  https://github.com/tokio-rs/tokio/blob/v0.1.x/tokio/examples/echo-udp.rs

# DNS: Tokio-based TCP server (raw future version)

## Objectives

- write a basic TCP server, using tokio and the dns crate

## Hints

- use the TCP echo server as an example
  https://github.com/tokio-rs/tokio/blob/v0.1.x/tokio/examples/echo.rs
- since the TCP DNS protocol is length-framed, you need two new methods of the dns crate:
  - dns::parse::parse_dns_tcp_packet
  - DnsPacket.serialize_tcp_to
- their interface is identical to the UDP counterparts

# DNS: Tokio-based TCP server (codec version)

## Objectives

- use the length_delimited codec and UDP parse/build methods from the dns crate to write a TCP DNS server

## Hints

- you need to restructure your code from the ground up
- instead of implementing a Future::poll method manually, you now need to:
  - create a codec instance from the TCP connection object

```
let dns_tcp_codec = length_delimited::Builder::new()
    .length_field_length(2)
    .new_framed(socket);
```

  - split the instance into a Stream (for reading frames) and a Sink (for writing frames)
  - implement your application logic with a combinator that maps a stream of requests to a stream

```
let (writer, reader) = self.socket.split();
```

    of responses
  - .forward() the resulting stream to the sink

```
let fut = reader
    .map(move |frame| match parse_dns_packet(&frame) {
        Ok((_, packet)) => {
            let mut out_buf = Vec::new();
            let resp = respond(packet, &addr);
            resp.serialize_to(&mut out_buf).and(Ok(out_buf))
        }
        Err(e) => Err(std::io::Error::new(
            std::io::ErrorKind::InvalidInput,
            format!("Malformed DNS query from {:?}: {:?}",
                    addr, e),
        )),
    })
    .filter_map(|resp| match resp {
        Ok(buf) => Some(buf.into()),
        Err(e) => {
            eprintln!("{}", e);
            None
        }
    })
    .forward(writer);
```

## Links

- https://docs.rs/tokio-io/0.1.5/tokio_io/codec/length_delimited/index.html

# DNS: combined TCP+UDP server

## Objectives

- use futures::lazy and tokio::spawn to run an application composed of two top-level futures

## Hints

- futures::lazy is a function that:
  - takes a closure which returns a future
  - calls that closure the first time it's poll()ed
- tokio::spawn runs a future on a tokio runtime
  - must be called while running under a tokio executor (i.e. from a future)
  - the spawned future:
    - must evaluate to () as there's nowhere to return the value
    - keeps the runtime alive until it finishes

```
tokio::run(lazy(|| {
    tokio::spawn(udp_server);
    tokio::spawn(tcp_server);
    Ok(())
}));
```