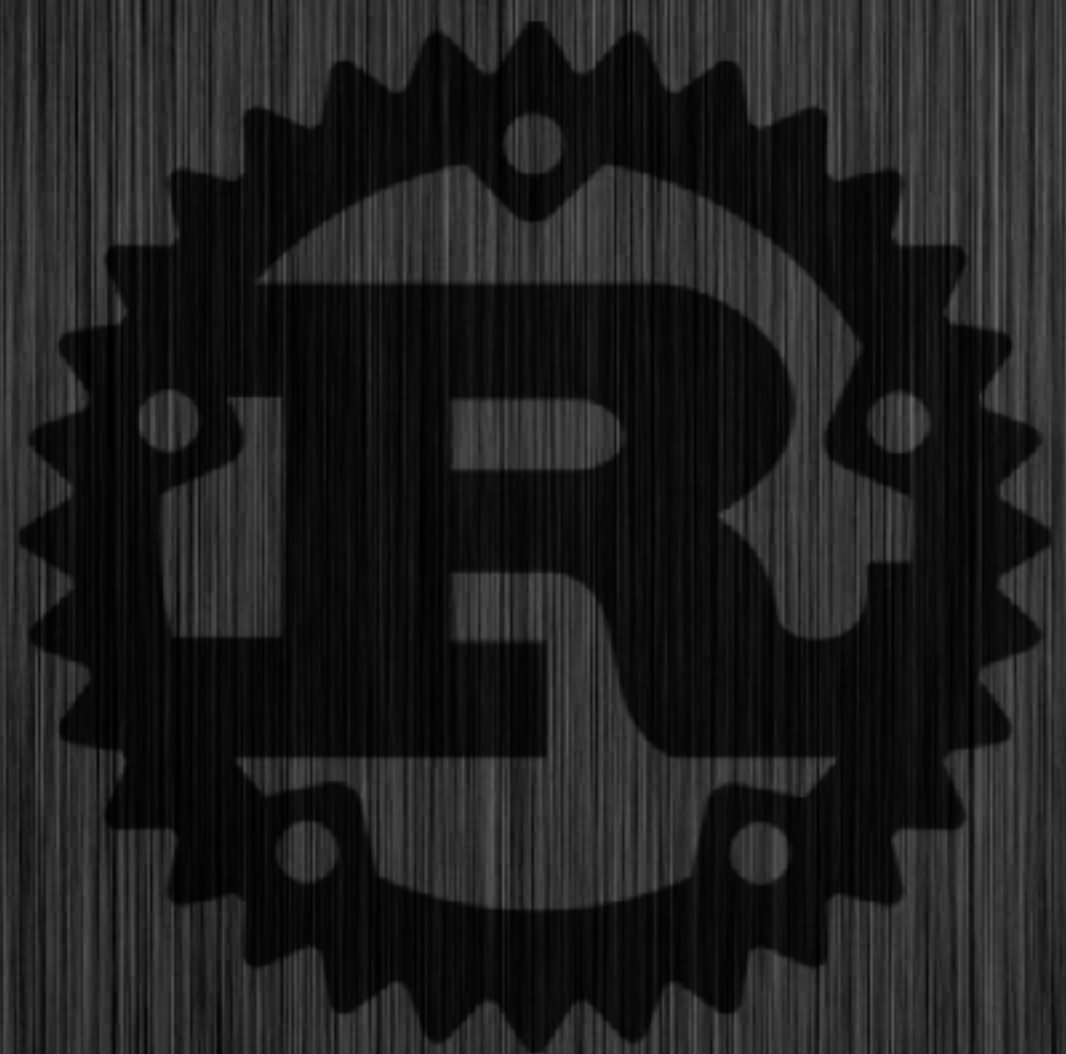
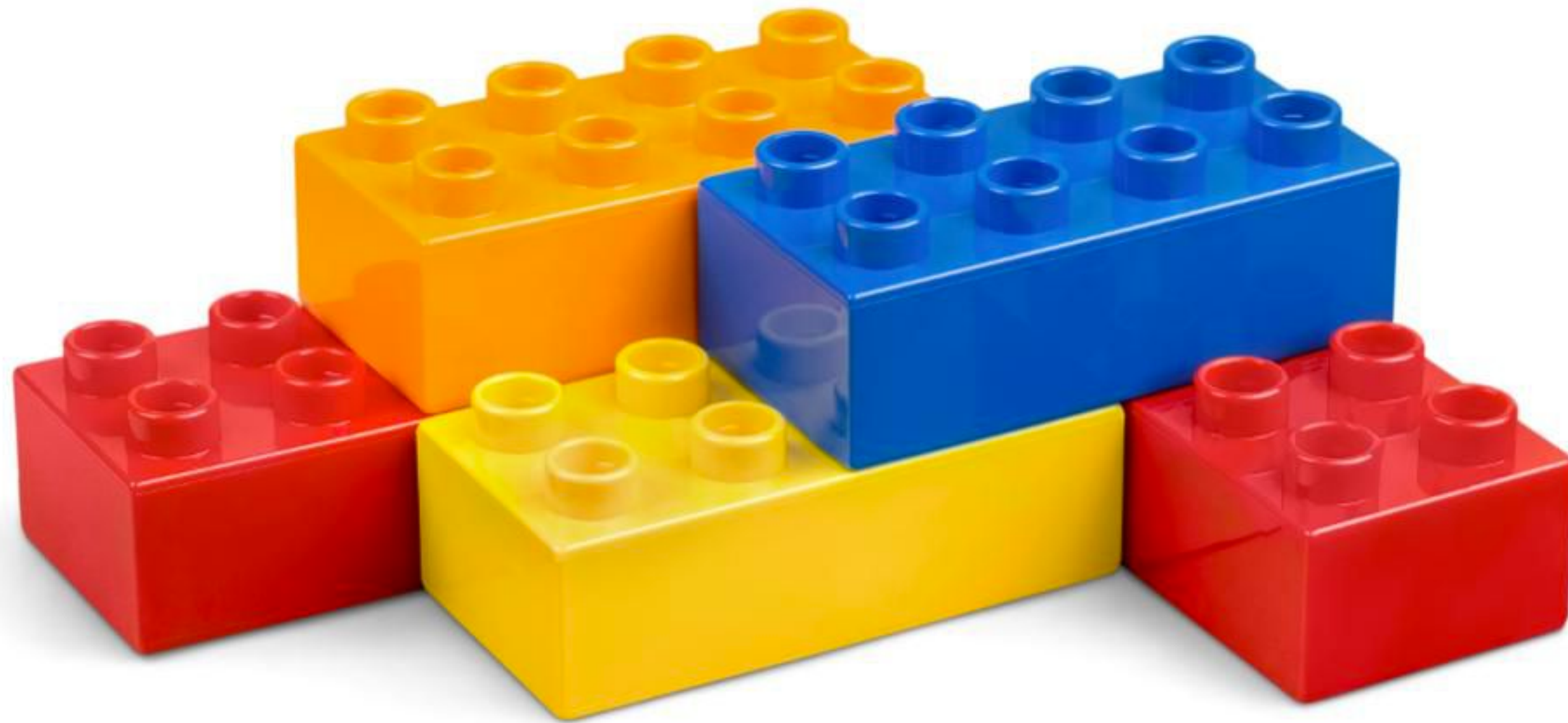




modules and visibility



modules and visibility

```
// external_module.rs
// external_module/mod.rs
mod external_module;
mod inline_module {
    use super::external_module::external_fn;

    pub(crate) fn internal_fn(i: u32) {
        external_fn(i+1);
    }
}
```

modules and visibility

```
mod error;
mod header;
mod label;
mod packet;
mod query;
mod resource_record;
mod response;
mod traits;

pub use error::*;
pub use packet::parse_dns_packet;

#[cfg(test)]
mod tests {
    use crate::parse::packet::parse_dns_packet;
    use crate::types::Opcode;
    use crate::types::Rcode;
    use crate::types::ResourceRecord;
    use std::net::Ipv4Addr;

    // ...
}
```

modules and visibility

C++ visibility rules

- **private: current class**
- **protected: current class and subclasses**
- **public: anybody**
- **friends may touch your privates**

modules and visibility

Rust visibility rules

- **private (default): current module**
- **pub(crate): current crate**
- **pub: anybody (has to be reachable from crate root)**
- **pub(self), pub(super), pub(in ...)**

modules and visibility

visibility rules apply to

- **types**
- **struct/enum/tuple struct fields**
- **functions**
- **methods in impl blocks**
- **imports (`use`)**
- **modules**

trait impls are always public

testing



testing

```
# cargo init --lib hello
  Created library package
# cd hello
# cargo test
  Compiling hello v0.1.0 (/src/rust/hello)
  Finished test [unoptimized + debuginfo] target(s) in 2.49s
  Running target/debug/deps/hello-0d7fd20a7eea7862

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

  Doc-tests hello

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

testing



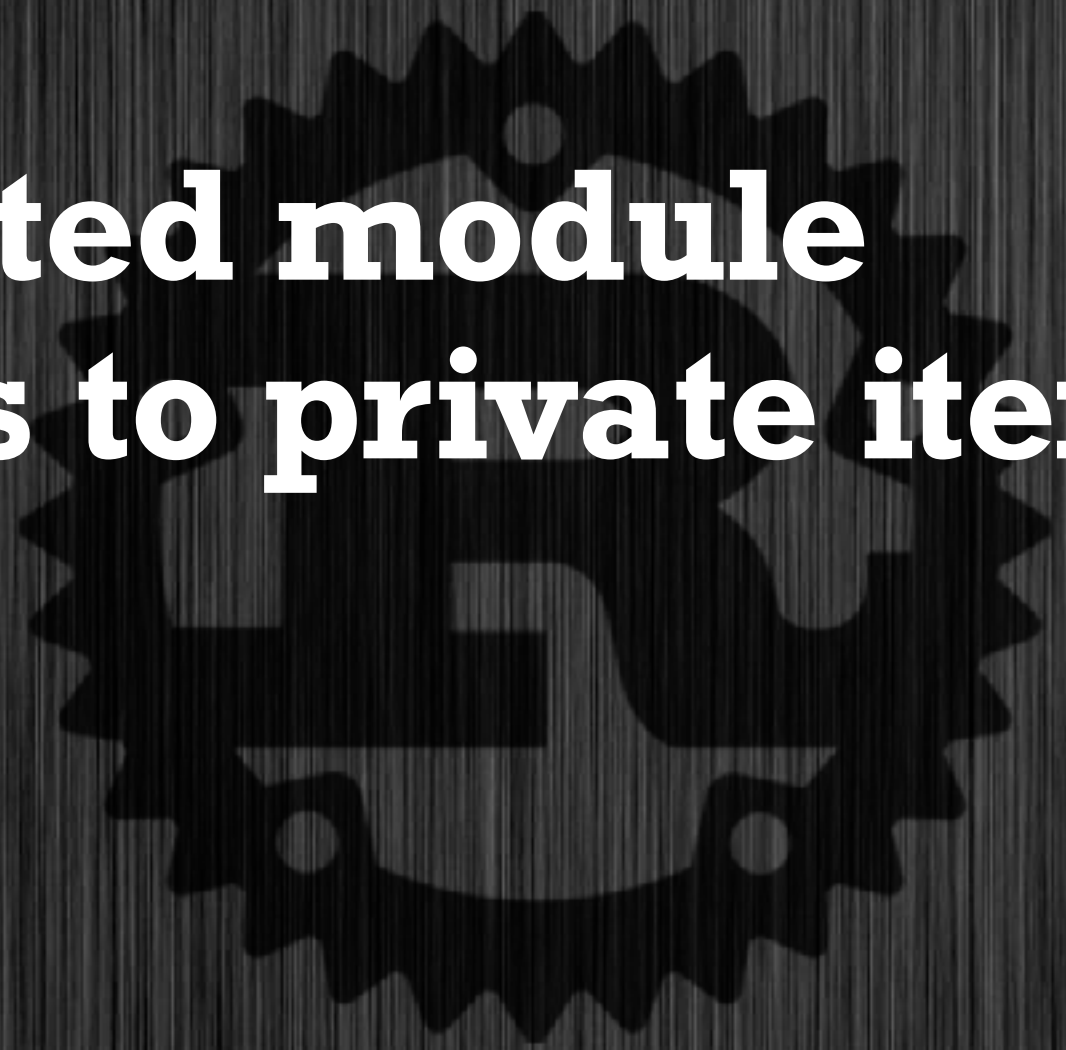
```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

testing basics

- **each function tagged with `#[test]` is a test case**
- **test failure indicated by panic**
 - **`panic!()`**
 - **`.unwrap()`**
 - **`assert!(boolean_expr [, "message"])`**
 - **`assert_eq!(a, b [, "message"])`**
 - **`assert_ne!(a, b [, "message"])`**
- **run using cargo test**

unit testing

- **#[cfg(test)]
mod tests {}**
 - **nested inside the tested module**
 - **tests have full access to private items**
 - **use `super::*`**



integration testing

- **tests/*.rs**
- **only have access to the public API**



documentation

```
//! This is a sample Rust library
//!
//! It exposes just a single function

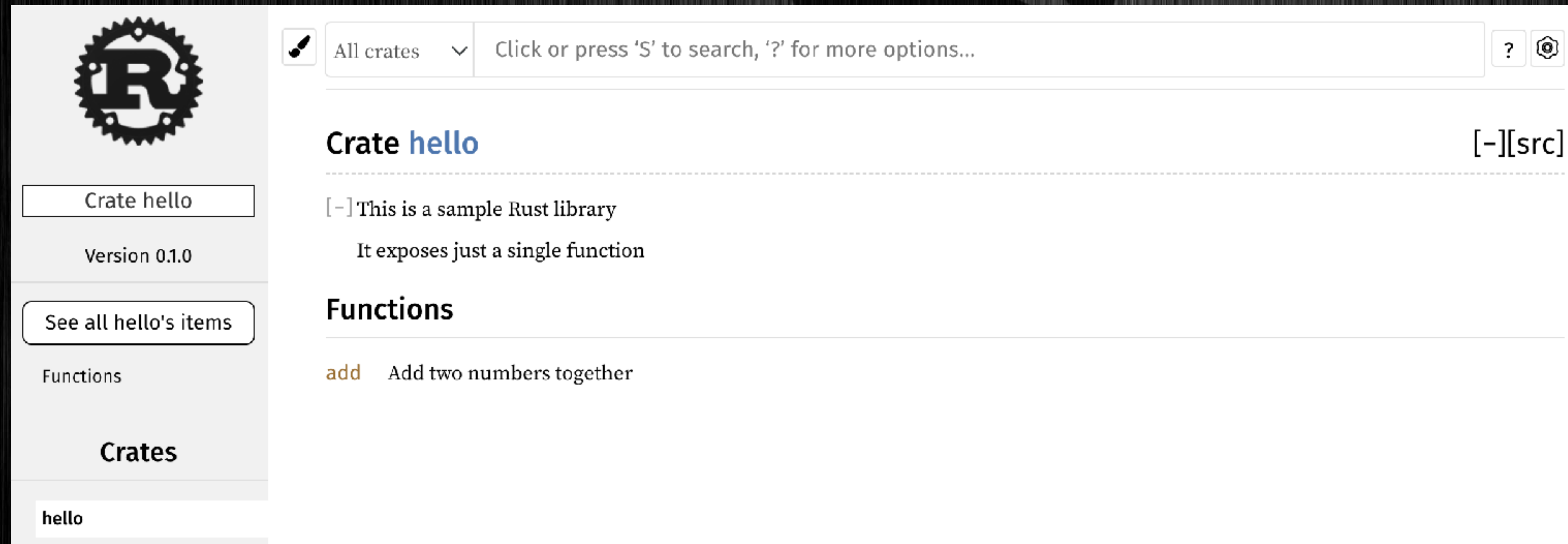
/// Add two numbers together
///
/// # Example
/// ```
/// let five = hello::add(3, 2);
/// assert_eq!(five, 5);
/// ```
pub fn add(a: u32, b: u32) -> u32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(add(2, 2), 4);
    }
}
```

documentation

- **cargo doc**
- **open target/doc/<crate>/index.html**



The screenshot shows the documentation page for the 'hello' crate on the Rust platform. The left sidebar contains the Rust logo, a button for 'Crate hello', the version '0.1.0', a button for 'See all hello's items', a 'Functions' link, and a 'Crates' link. The main content area has a search bar at the top with a dropdown menu set to 'All crates' and a search prompt. Below the search bar, the crate name 'hello' is displayed with links for '[-]' and '[src]'. The description states 'This is a sample Rust library' and 'It exposes just a single function'. Under the 'Functions' section, the 'add' function is listed with the description 'Add two numbers together'.



Crate hello

Version 0.1.0

See all hello's items

Functions

Crates

hello

 All crates ▾ Click or press 'S' to search, '?' for more options...  

Crate **hello** [-] [src]

[-] This is a sample Rust library

It exposes just a single function

Functions

add Add two numbers together

documentation

- **cargo doc**
- **open target/doc/<crate>/index.html**



The screenshot shows the Rust documentation website. On the left is a sidebar with the Rust logo, a search bar containing 'hello', and a list of crates and functions. The 'hello' crate is selected, and the 'add' function is highlighted. The main content area shows the function signature 'Function hello::add' with links for source code and documentation. Below the signature is the function's implementation in Rust code. Further down is an 'Example' section with a code snippet demonstrating the use of the 'add' function.

 All crates Click or press 'S' to search, '?' for more options... ? 

Function hello::add [\[-\]](#) [\[src\]](#)

```
pub fn add(a: u32, b: u32) -> u32
```

[\[-\]](#) Add two numbers together

Example

```
let five = hello::add(3, 2);
assert_eq!(five, 5);
```

documentation

```
$ cargo test
  Finished test [unoptimized + debuginfo] target(s) in 0.00s
  Running unittests (target/debug/deps/hello-0af4a5d0fd4fc68d)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

    Doc-tests hello

running 1 test
test src/lib.rs - add (line 8) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.15s
```

A detailed illustration of a dragon's head and neck, breathing a powerful stream of fire. The dragon has orange and yellow scales, a red tongue, and sharp teeth. The fire is bright yellow and orange, with a small white flame at the end of the stream. The background is a dark, cloudy sky.

**rule 1
of unsafe:
don't**

unsafe

- **dereference raw pointers**
- **call unsafe functions**
- **implement unsafe traits**
- **mutate statics**
- **access fields of unions**



unsafe

- **FFI**
non-Rust code can't maintain Rust guarantees
- **bare-metal development**
e.g. memory-mapped I/O
- **data structures**
you will probably need a raw pointer eventually
- **performance hacks**
profile to make sure it's worth it



unsafe

- **compiler can't help you there**
- **define a safe/unsafe boundary in your code
(encapsulate all unsafety in a function/module)**
- **bugs in unsafe code can manifest in safe code**
- **not a way to bypass the borrow checker
(e.g. multiple mutable references are still UB)**

unsafe

- **if you can prove safety in all cases
make a safe function with an unsafe block**
- **if you cannot, make the function unsafe
and document the contract**



<https://doc.rust-lang.org/nomicon/>

<https://os.phil-opp.com/>

unsafe: dereference raw pointers

```
use std::ptr;

// Note: This definition is naive. See the chapter on implementing Vec.
pub struct Vec<T> {
    ptr: *mut T,
    len: usize,
    cap: usize,
}

// Note this implementation does not correctly handle zero-sized types.
// See the chapter on implementing Vec.
impl<T> Vec<T> {
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap {
            // not important for this example
            self.reallocate();
        }
        unsafe {
            ptr::write(self.ptr.offset(self.len as isize), elem);
            self.len += 1;
        }
    }
}
```

unsafe: dereference raw pointers

```
fn make_room(&mut self) {  
    // grow the capacity  
    self.cap += 1;  
}
```

```
// Note this implementation does not correctly handle zero-sized types.  
// See the chapter on implementing Vec.  
impl<T> Vec<T> {  
    pub fn push(&mut self, elem: T) {  
        if self.len == self.cap {  
            // not important for this example  
            self.reallocate();  
        }  
        unsafe {  
            ptr::write(self.ptr.offset(self.len as isize), elem);  
            self.len += 1;  
        }  
    }  
}
```

unsafe: call unsafe functions

```
pub unsafe extern "C" fn lchown(  
    path: *const c_char,  
    uid: uid_t,  
    gid: gid_t  
) -> c_int
```

```
fn lchown<P: AsRef<Path>>(p: P, uid: usize, gid: usize) -> std::io::Result<()> {  
    let p = p.as_ref().as_os_str();  
    let c_path = CString::new(p.as_bytes())?;  
  
    let ret =  
        unsafe { libc::lchown(c_path.as_ptr(), uid as libc::c_uint, gid as libc::c_uint) };  
    if ret == -1 {  
        Err(std::io::Error::last_os_error())  
    } else {  
        Ok(())  
    }  
}
```

unsafe: implement unsafe traits

extra assumptions that the compiler can't verify

```
pub unsafe trait Searcher<'a> {  
    fn haystack(&self) -> &'a str;  
    fn next(&mut self) -> SearchStep;  
  
    fn next_match(&mut self) -> Option<(usize, usize)> { ... }  
    fn next_reject(&mut self) -> Option<(usize, usize)> { ... }  
}
```

```
unsafe impl<'a> Searcher<'a> for MySearcher<'a> {  
    // ...  
}
```

unsafe: watch the world burn



```
pub fn get_mut(&self) -> &mut HttpInnerMessage {  
    let r: &HttpInnerMessage = self.0.as_ref().unwrap().as_ref();  
    unsafe { &mut *(r as *const _ as *mut _) }  
}
```



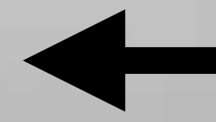
FFI

real-world FFI

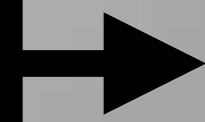
- **libc crate**
- **foo-sys crates**
sources of libfoo usually bundled
- **example:**
zstd-sys (raw FFI interface)
zstd-safe (safe but inconvenient abstraction)
zstd (safe and convenient)

FFI from scratch

Rust



C



C++

FFI: linking with external libraries

```
#[link(name = "foo")]  
extern "C" pub fn foo_open(...);
```

three more string types

- **std::ffi::CString**
owned NUL-terminated string
not FFI-safe
- **std::ffi:CStr**
borrowed NUL-terminated string
not FFI-safe
- ***const std::os::raw::c_char**
compatible with C ``const char*``

three more string types

```
use std::ffi::{CString, NulError};

fn c_strlen_ok(rust_str: &str) -> Result<usize, NulError> {
    let c_string = CString::new(rust_str)?;
    Ok(unsafe { libc::strlen(c_string.as_ptr()) as usize })
}

fn c_strlen_bad(rust_str: &str) -> Result<usize, NulError> {
    Ok(unsafe { libc::strlen(CString::new(rust_str)?.as_ptr()) as usize })
}
```

wrapping libsnappy

<https://doc.rust-lang.org/nomicon/ffi.html>

```
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

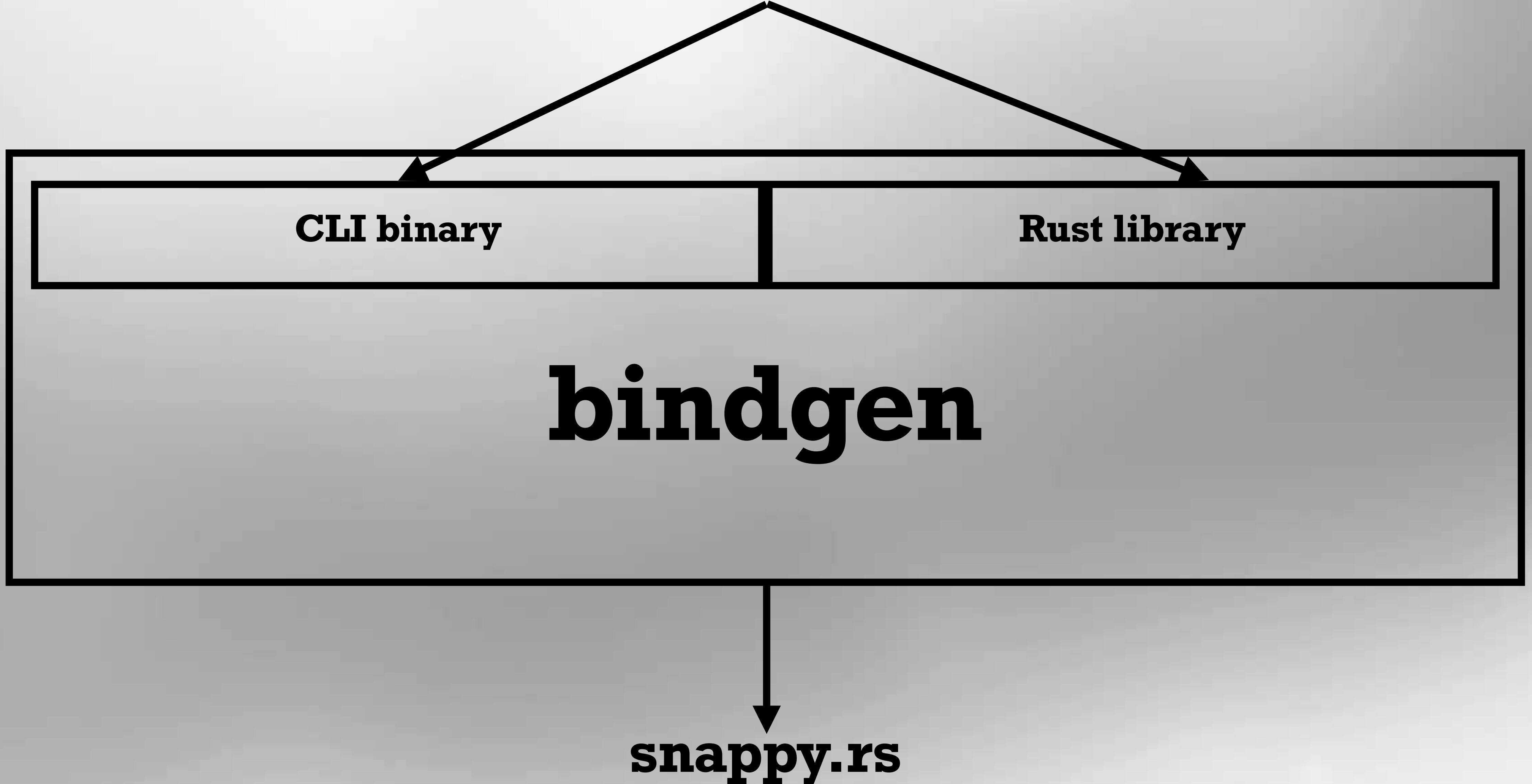
snappy-c.h

CLI binary

Rust library

bindgen

snappy.rs



bindgen /usr/include/snappy-c.h

```
pub const snappy_status_SNAPPY_OK: snappy_status = 0;
pub const snappy_status_SNAPPY_INVALID_INPUT: snappy_status = 1;
pub const snappy_status_SNAPPY_BUFFER_TOO_SMALL: snappy_status = 2;
pub type snappy_status = ::std::os::raw::c_uint;
extern "C" {
    pub fn snappy_compress(
        input: *const ::std::os::raw::c_char,
        input_length: size_t,
        compressed: *mut ::std::os::raw::c_char,
        compressed_length: *mut size_t,
    ) -> snappy_status;
}
extern "C" {
    pub fn snappy_uncompress(
        compressed: *const ::std::os::raw::c_char,
        compressed_length: size_t,
        uncompressed: *mut ::std::os::raw::c_char,
        uncompressed_length: *mut size_t,
    ) -> snappy_status;
}
extern "C" {
    pub fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}
```

FFI: build.rs

compile-time code generation
extra compiler/linker options

```
[package]
name = "foo-sys"
links = "foo"
build = "build.rs"
```

```
fn main() {
    println!("cargo:rustc-link-search={}", ::std::env::var("LIBF00_DIR").unwrap());
    println!("cargo:rustc-link-lib=static=foo");
}
```

kornel.ski/rust-sys-crate

CXX.rs

passing data across FFI boundary

- **Vec::as_ptr(), Vec::size()**
 - **&foo as *const _**
 - **#[repr(C)]**
struct Foo { ... }
 - **Box::into_raw(), Box::from_raw()**
pass ownership to/from FFI code
- **not** malloc/free compatible**

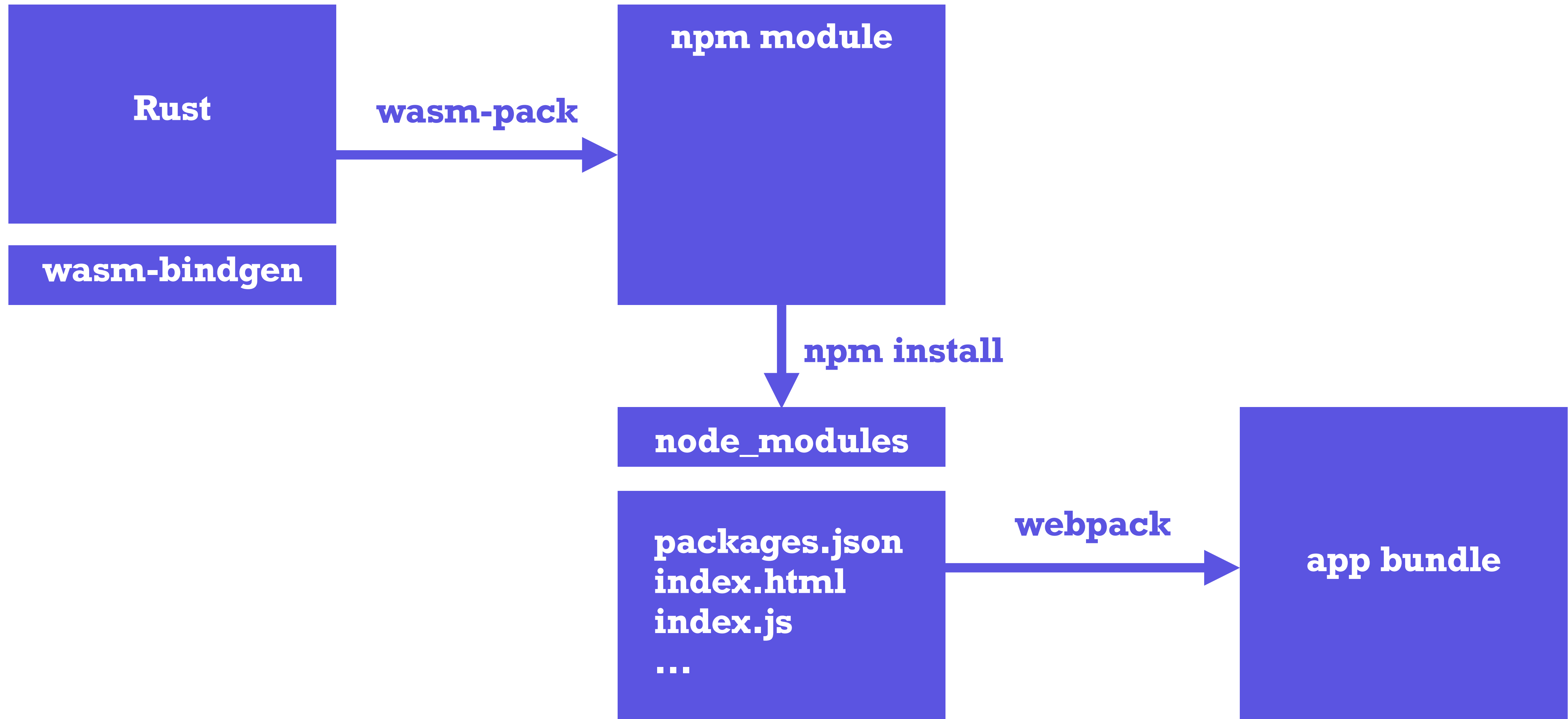


Rust + WebAssembly

Rust + WebAssembly

- **disclaimer: not a web developer**
- **javascript is what happens to other people**
- **<https://rustwasm.github.io/docs/book/>**

Rust + WebAssembly



Rust + WebAssembly

- **wasm32-unknown-unknown target**
- **wasm-pack installs it if not present**

Rust + WebAssembly

- **wasm-bindgen crate**
- **`#[wasm_bindgen]` annotation for**
 - **types**
 - **impl blocks**
 - **functions**
- **import from JS**
- **export to JS**

Rust + WebAssembly

```
#[wasm_bindgen]
pub struct Game(game::Game);

// ...

#[wasm_bindgen]
impl Game {
    // ...

    pub fn game_over(&self) -> bool {
        match self.0.state() {
            game::State::Player0Move | game::State::PlayerXMove => false,
            _ => true,
        }
    }

    // ...
}
```

Rust + WebAssembly

- **wasm-pack build**

```
pkg
├── README.md
├── package.json
├── tictactoe.d.ts
├── tictactoe.js
├── tictactoe_bg.js
├── tictactoe_bg.wasm
└── tictactoe_bg.wasm.d.ts
```

Rust + WebAssembly

- **TypeScript declaration file**

```
export class Game {  
    free(): void;  
    static new(): Game;  
    do_move(row: number, column: number): boolean;  
    do_ai_move(difficulty: number): boolean;  
    get_board(): Uint8Array;  
    get_state(): string;  
    game_over(): boolean;  
}
```

- **(also includes generated documentation)**

Rust + WebAssembly

- **npm init wasm-app APP_NAME**
- **<https://github.com/rustwasm/create-wasm-app>**

Rust + WebAssembly

- **npm install**

```
"dependencies": {  
  "tictactoe": "file:../tictactoe/pkg"  
},
```

Rust + WebAssembly

- **index.js**

```
import { Game } from "tictactoe";

const restart = () => {
  game = Game.new();
  render();
}

const move = (row, column) => {
  if(game.do_move(row, column)) {
    render();
  }
}

// ...

let game;
init();
restart();
```

Rust + WebAssembly

- **npm run start**

← → ↻ ⓘ localhost:8080

Player O moves

.	.	.
.	X	.
X	.	O

AI level: Easy ▼ Move

writing web applications with



ACTIX

<https://actix.rs/>

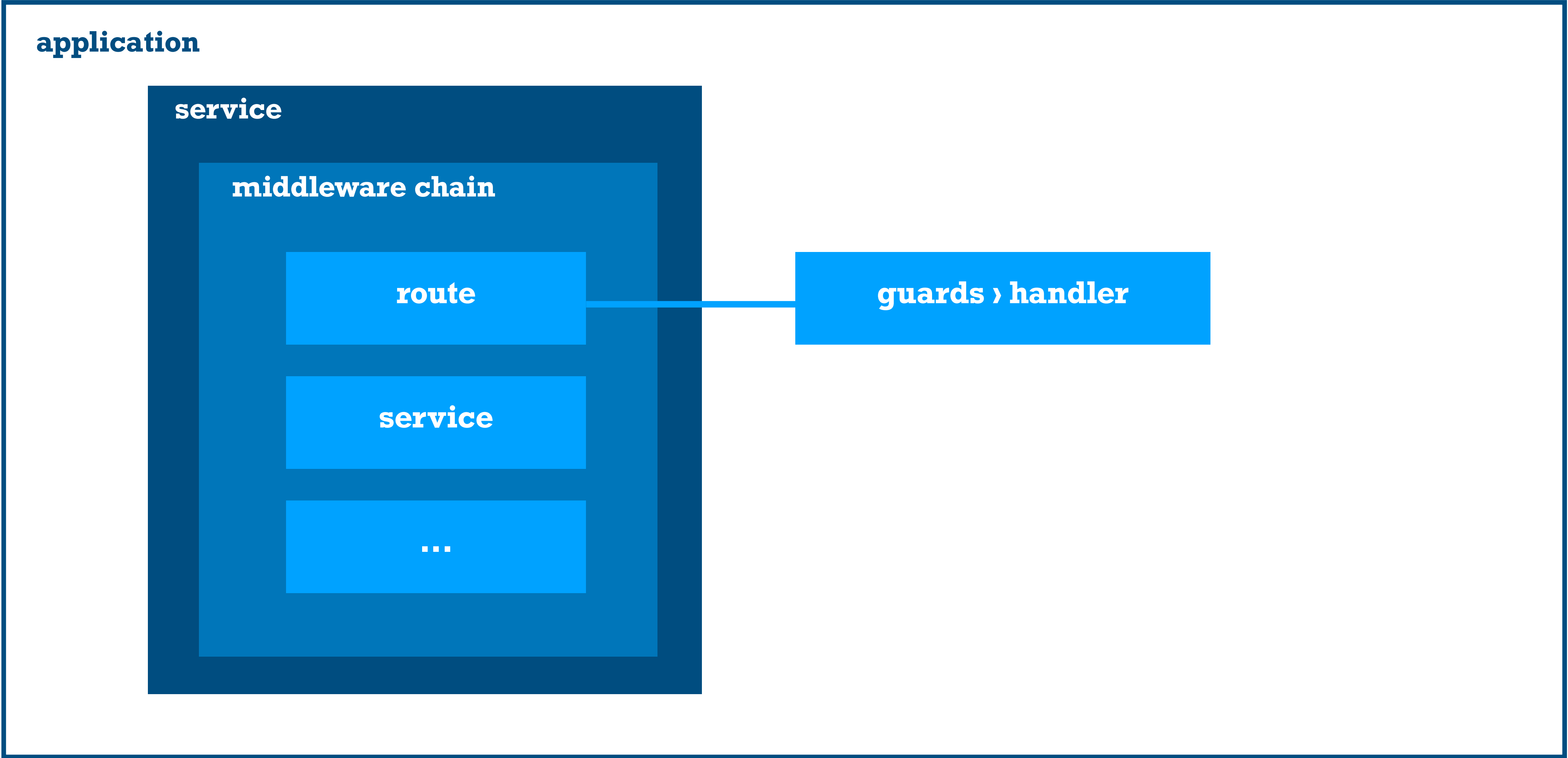
Why Actix?

- **not the only framework**
- **<https://www.arewewebyet.org/>**

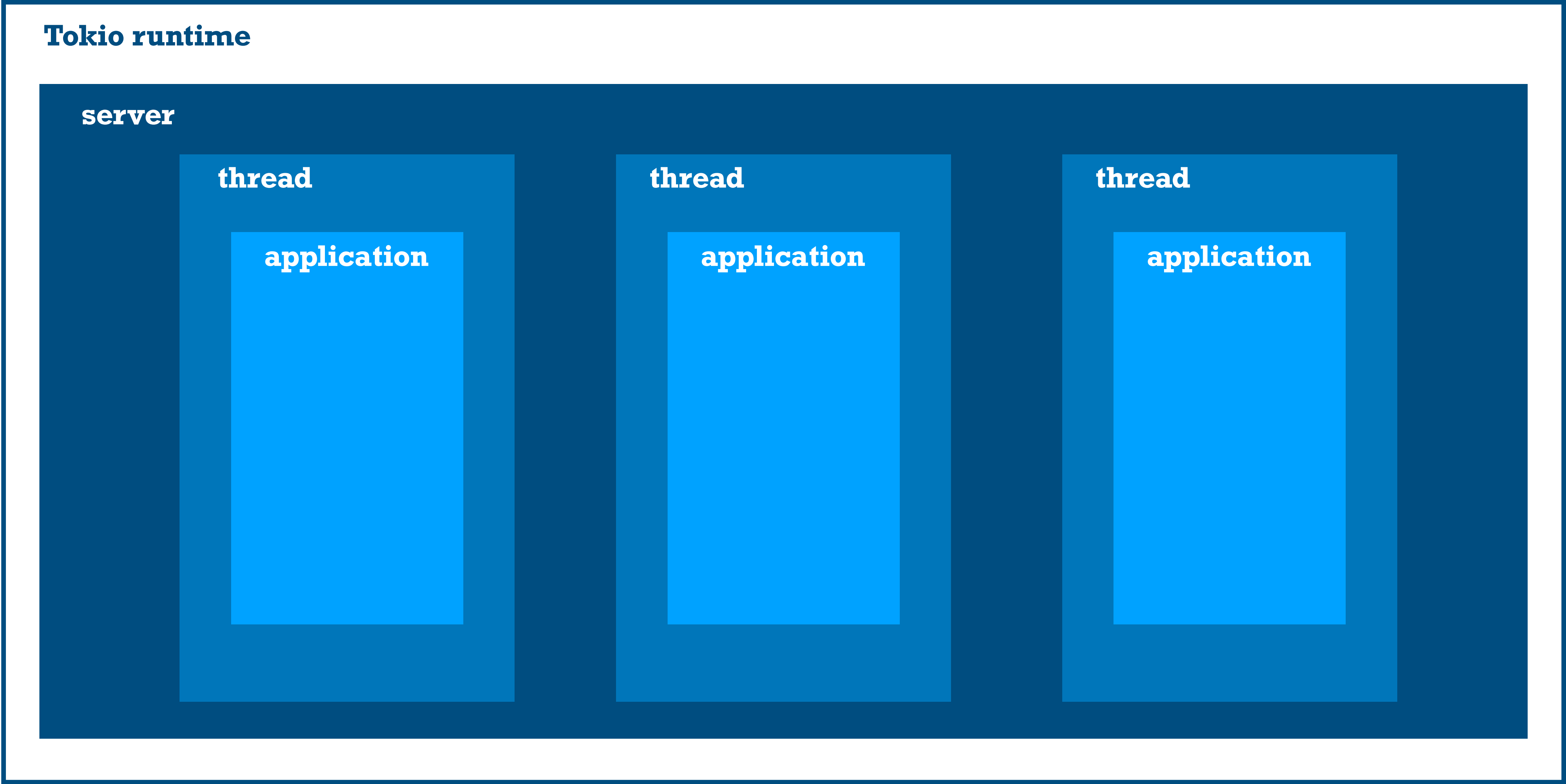
Why Actix?

- **fast**
- **feature-rich**
 - **fully async**
 - **TLS**
 - **HTTP/2, websockets**
- **works with stable Rust**
- **decent usability & ecosystem**

Actix web apps: code view



Actix web apps: runtime view



A bare-bones server

```
use actix_web::{web, App, HttpRequest, HttpServer, Responder};

async fn greet(req: HttpRequest) -> impl Responder {
    let name = req.match_info().get("name").unwrap_or("World");
    format!("Hello {}!", &name)
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(greet))
            .route("/{name}", web::get().to(greet))
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

Anatomy of a request handler

a handler is:

- **an async function**
- **taking zero or more parameters (extractors)**
- **returning a type that implements Responder**

```
async fn capture_event(evt: web::Json<Event>) -> impl Responder {  
    let new_event = store_in_db(evt.timestamp, &evt.kind, &evt.tags);  
    format!("got event {}", new_event.id.unwrap())  
}
```

Extractors

- **a type implementing FromRequest**
- **i.e. some data extracted from the request**
 - **path fragments**
 - **query string**
 - **body (as JSON)**
 - **body (as a form)**
 - **etc.**

Extractors

- **path fragments**
- **query string**
- **body (as JSON)**
- **body (as a form)**

Path<T>
Query<T>
Json<T>
Form<T>

what's the T?

Extractors

- **all extracted data is strongly typed**
- **basic validation (e.g. required fields) already built in**
- **uses serde underneath**

Serde deserves a little detour

- **generic serialization/deserialization library**
- **how: actual formats provided by external crates**
 - **serde_json (JSON)**
 - **rmp_serde (MessagePack)**
 - **many more**
- **what: any type**
 - **must implement `serde::Serialize`**
 - **and/or `serde::Deserialize`**
 - **custom derive attribute for great usability**

Serde deserves a little detour

```
#[derive(Serialize, Deserialize)]
struct Person {
    name: String,
    age: u8,
    phones: Vec<String>,
}

let p: Person = serde_json::from_str(data)?; // Deserialize
let j = serde_json::to_string(&p)?;          // Serialize
```

<https://serde.rs/>

App data: a different type of extractor

- **App.data(anything)**
- **available as an extractor (Data<T>)**
 - **passing global configuration to handlers**
 - **shared mutable state (Data<Mutex<T>>)**

Responders

can be converted to an HTTP response

- **built in implementations**
 - **String**
 - **Bytes, Vec<u8>**
 - **Json<T>**
 - **Option<impl Responder>**
 - **Result<impl Responder, impl Into<Error>>**
- **custom implementations**

Routes

route = (0 or more) guards + one handler

impl Guard

- **fn check(req: &GuardContext) -> bool**

built-in guards:

- **HTTP methods**
- **matching headers**
- **impl Fn(&GuardContext) -> bool**

Error handling

impl ResponseError

fn status_code(&self) -> StatusCode { ... }

fn error_response(&self) -> HttpResponse { ... }

default impls:

status code = 500

body from Display trait

Testing

unit testing individual handlers

- **handlers are functions so just call them**
 - **annotating with e.g. `#[get("/url")]` converts the function into a service**
- **`test::TestRequest` lets you create mock HTTP requests**

Testing

integration testing

- test the whole stack (routing, middleware, guards...)
- `let app = test::init_service(App::new()...)`
- `let req = test::TestRequest...`
- `app.call(&req)`
- useful helpers
 - `test::read_response`
 - `test::read_response_json`

