DEVELOPING A
FASTER AND SAFER WEB
WITH RUST

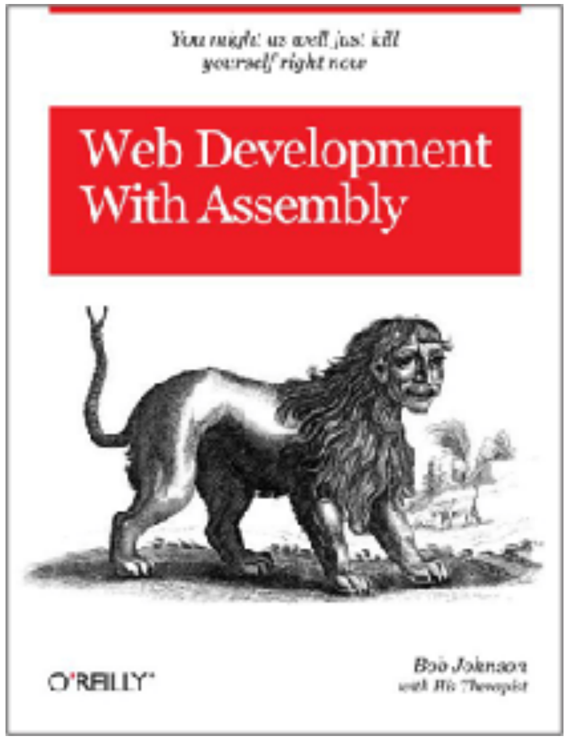
Stanko Krtalic Rusendic

github.com/Stankec
@monorkin

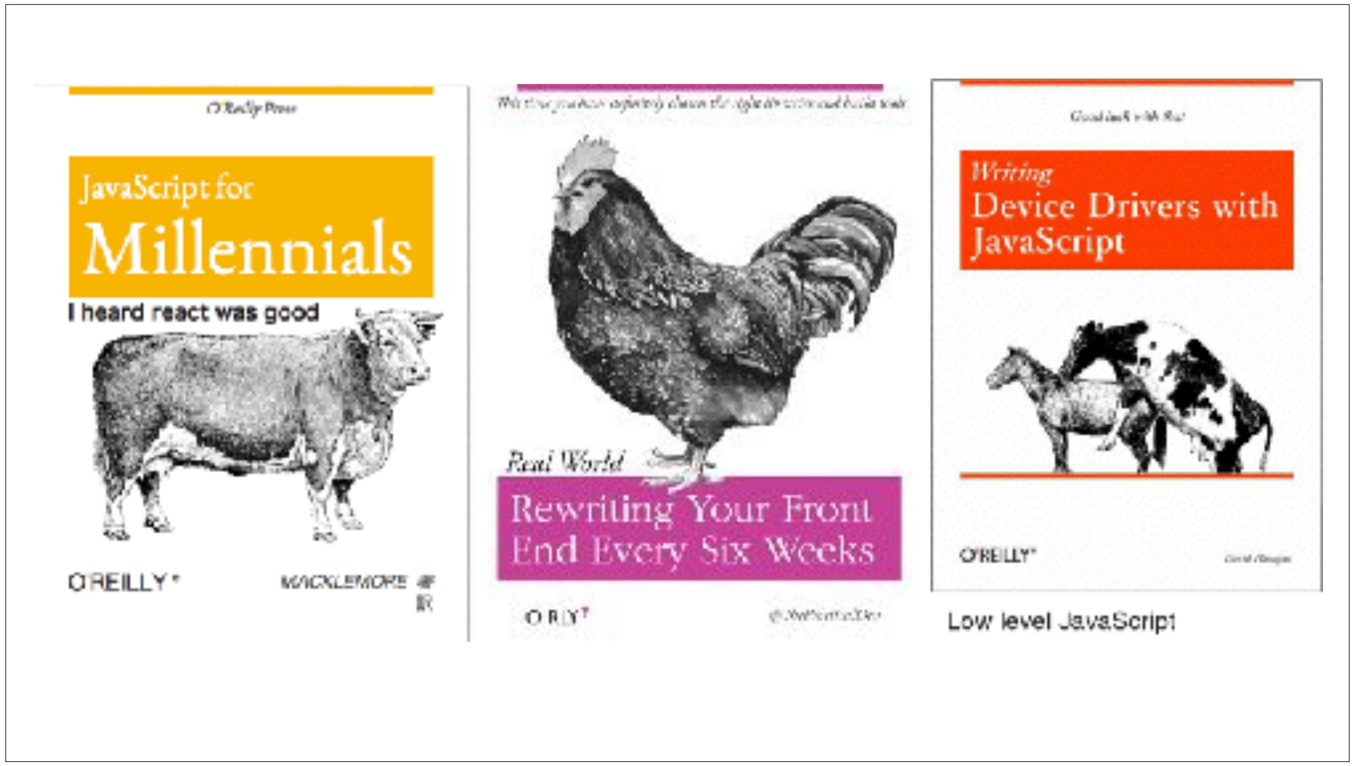
01

OMG RUST

When I tell people that I write web applications in rust I usually get weird looks.

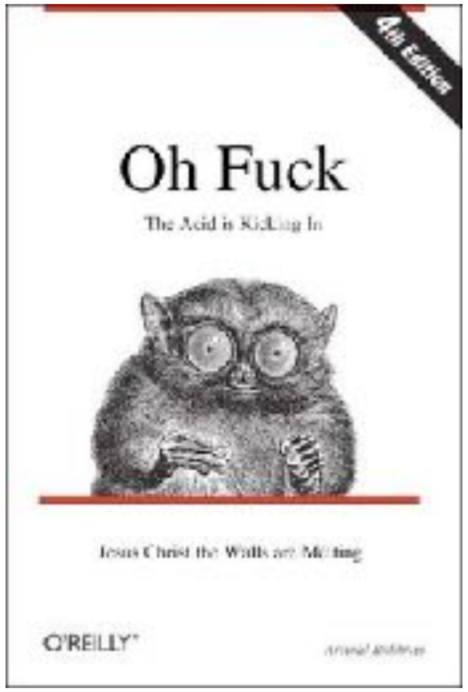

You might as well just kill yourself right now
Web Development With Assembly
O'REILLY
Bob Johnson
with His Therapist

People mostly compare it to doing web development in assembly.


O'Reilly Press
JavaScript for
Millennials
I heard react was good
O'REILLY
MACKLEMORE
This time you have definitely chosen the right libraries and build tools
Real World
Rewriting Your Front End Every Six Weeks
O RLY?
Good luck with that
Writing
Device Drivers with JavaScript
O'REILLY

Low level JavaScript

But I can assure you it’s better than doing Javascript



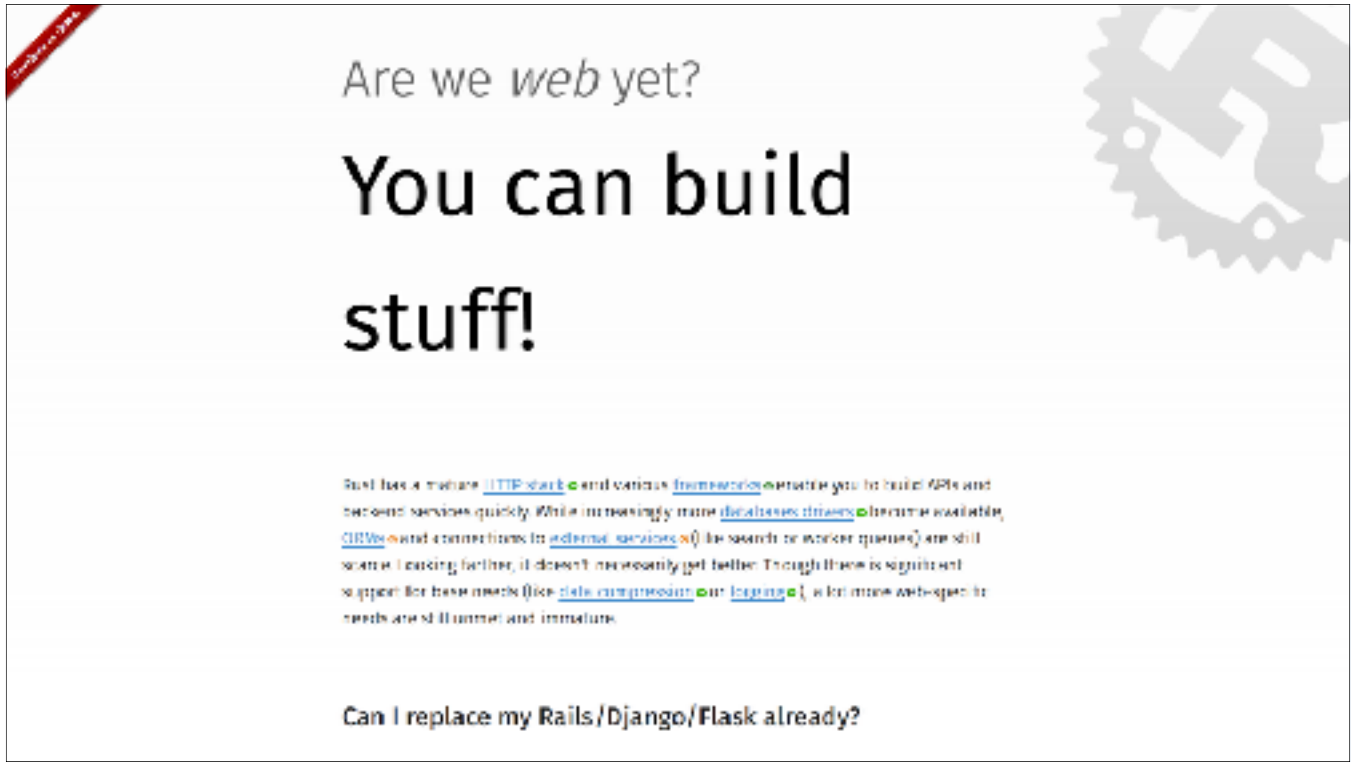
But worse than doing drugs, saidly.

Safety

Not to bore you to death with everything. Rust gives you safety. A program written in Rust gan only have logical errors, in the sense that your algorythm is wrong. There can't be any race conditions, isecure data acces, volotile data, and what not.
The problem is that the web consists of a lot of moving parts. That's why we need to find a way to interact with it in a safe manner.

02

So this talks is mostly going to answer the question of can you build web apps with Rust, how complicated it is and what are the benefits



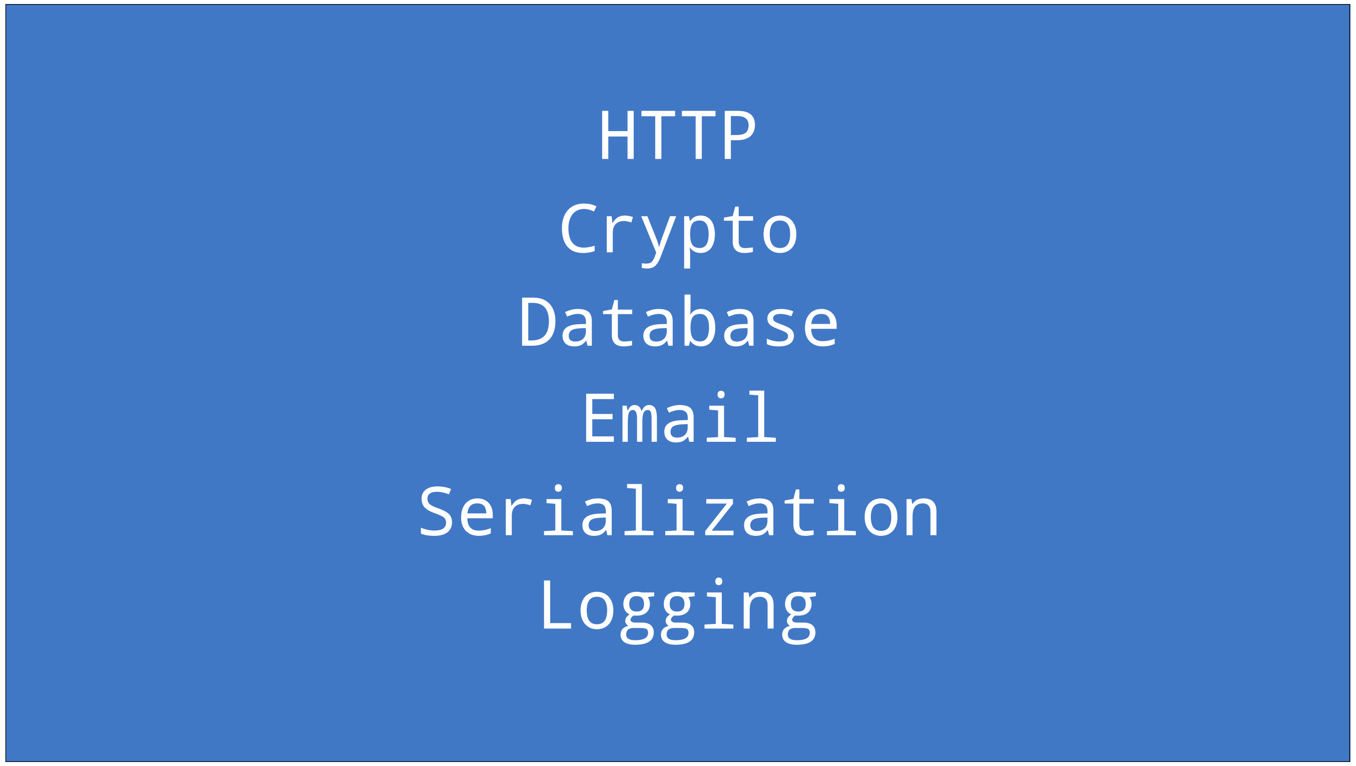
Are we *web* yet?

You can build stuff!

Rust has a mature HTTP stack and various frameworks enable you to build APIs and backend services quickly. While increasingly more databases drivers become available, ORMs and connections to external services (like search or worker queues) are still scarce. Looking farther, it doesn't necessarily get better. Though there is significant support for base needs (like data compression or logging), a lot more web-specific needs are still unmet and immature.

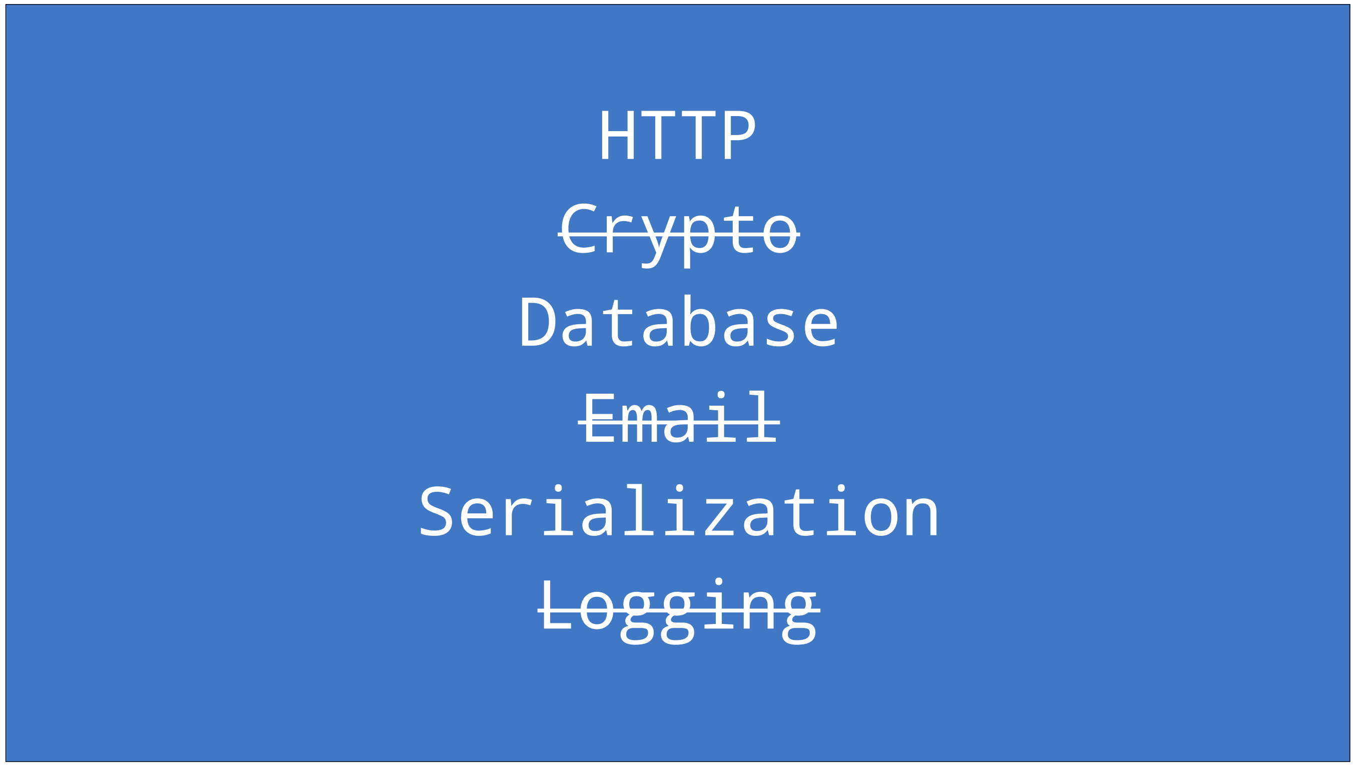
Can I replace my Rails/Django/Flask already?

There is this site - arewewebyet.com
It will tell you at any point in time if you can build web applications with Rust.
It also has a comrehansive list of available libraries for different tasks so it’s worth checking out if you want to venture down my path and build a web app in Rust.



HTTP
Crypto
Database
Email
Serialization
Logging

It outlines 6 main categories.
HTTP stack - for buildinng server, processing requests and buiolding responses
Crypto - for securing data
Database - for storing data
Email - For indirect communication with the user
Serialization - For building services
Logging - so that you have a clue about what’s happening



HTTP

~~Crypto~~

Database

~~Email~~

Serialization

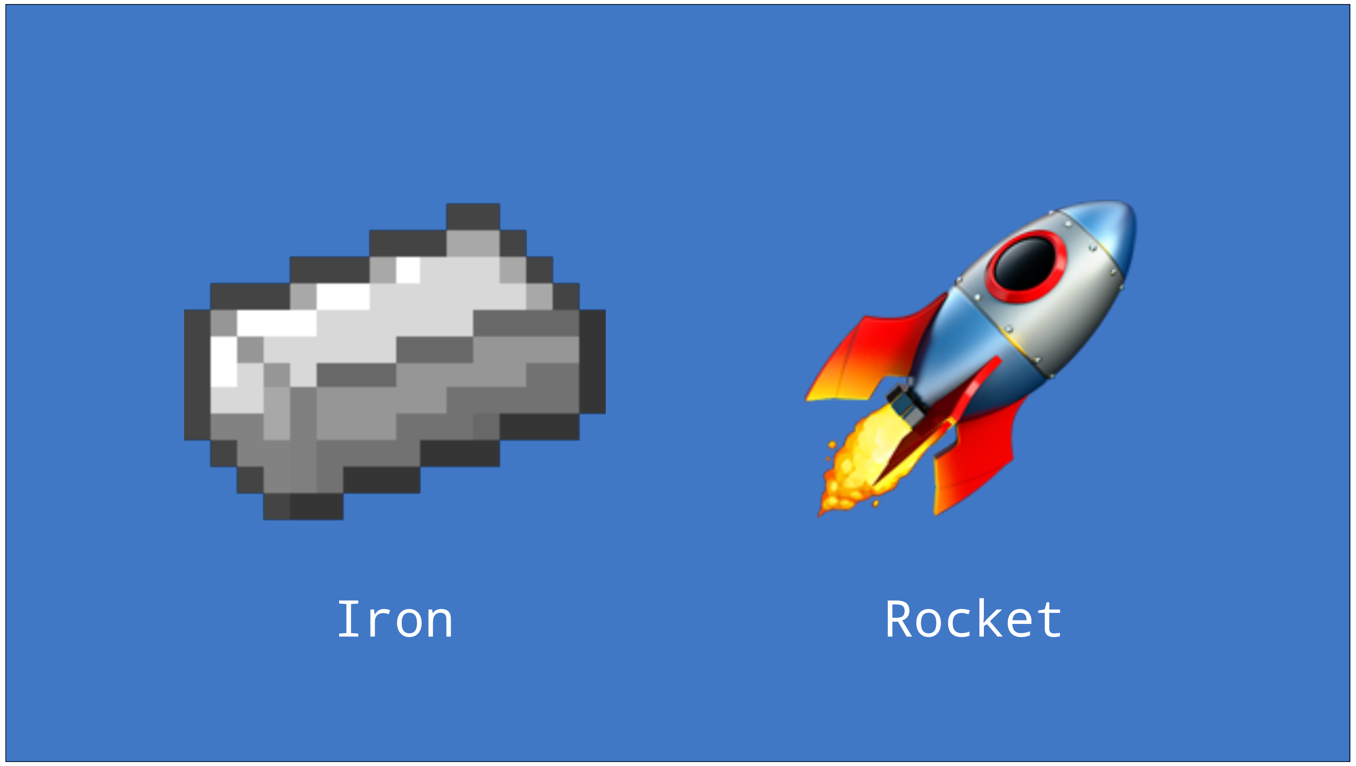
~~Logging~~

In this talk I’ll omit the Crypto and Email part. Mostly because you will rarely use crypto libraries direclty, and most people use third party services for email related stuff. And they can be a huge time sink to explain. Also logging is really trivial.

03

HTTP

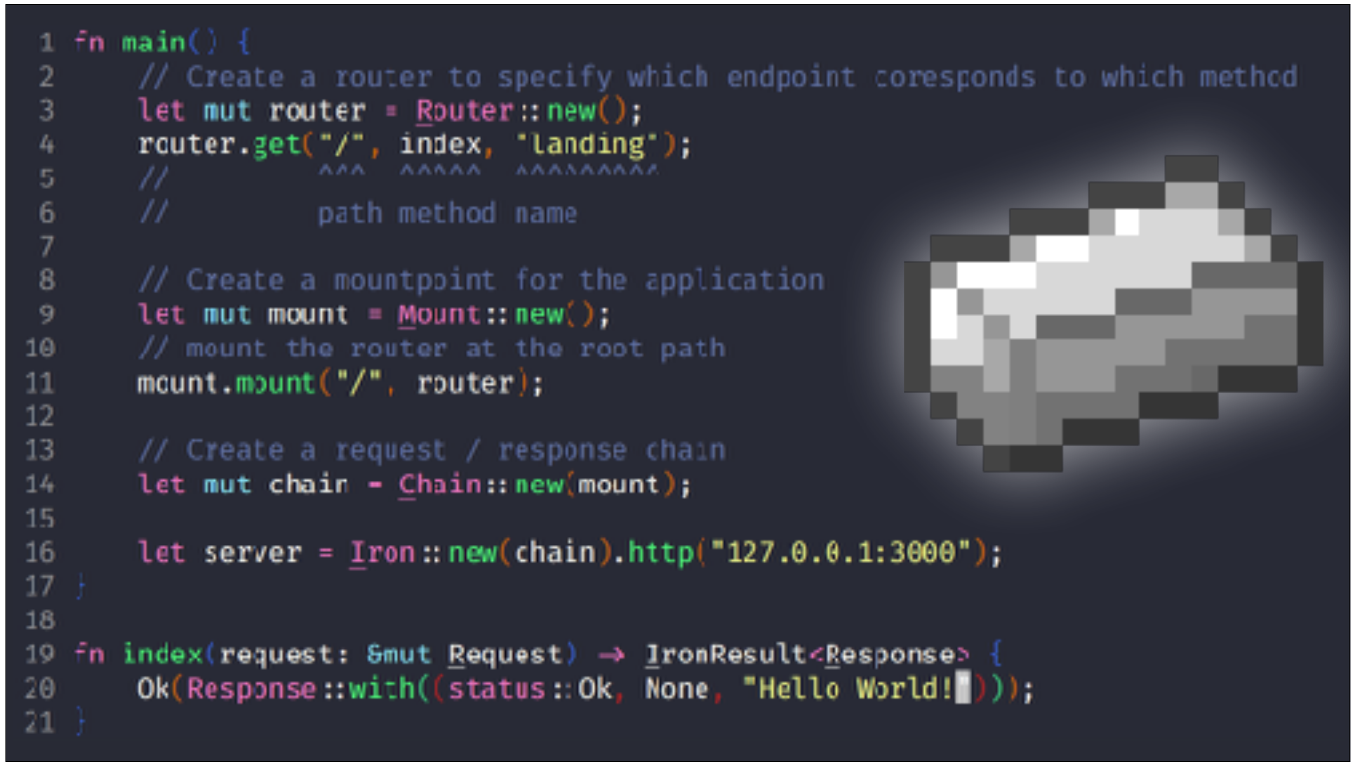
Let’s start with HTTP.
To make a web application we need a server.
while you could create a TCP event loop and implement things your self I’d advise you to take a look at these two fantastic projects.



Iron

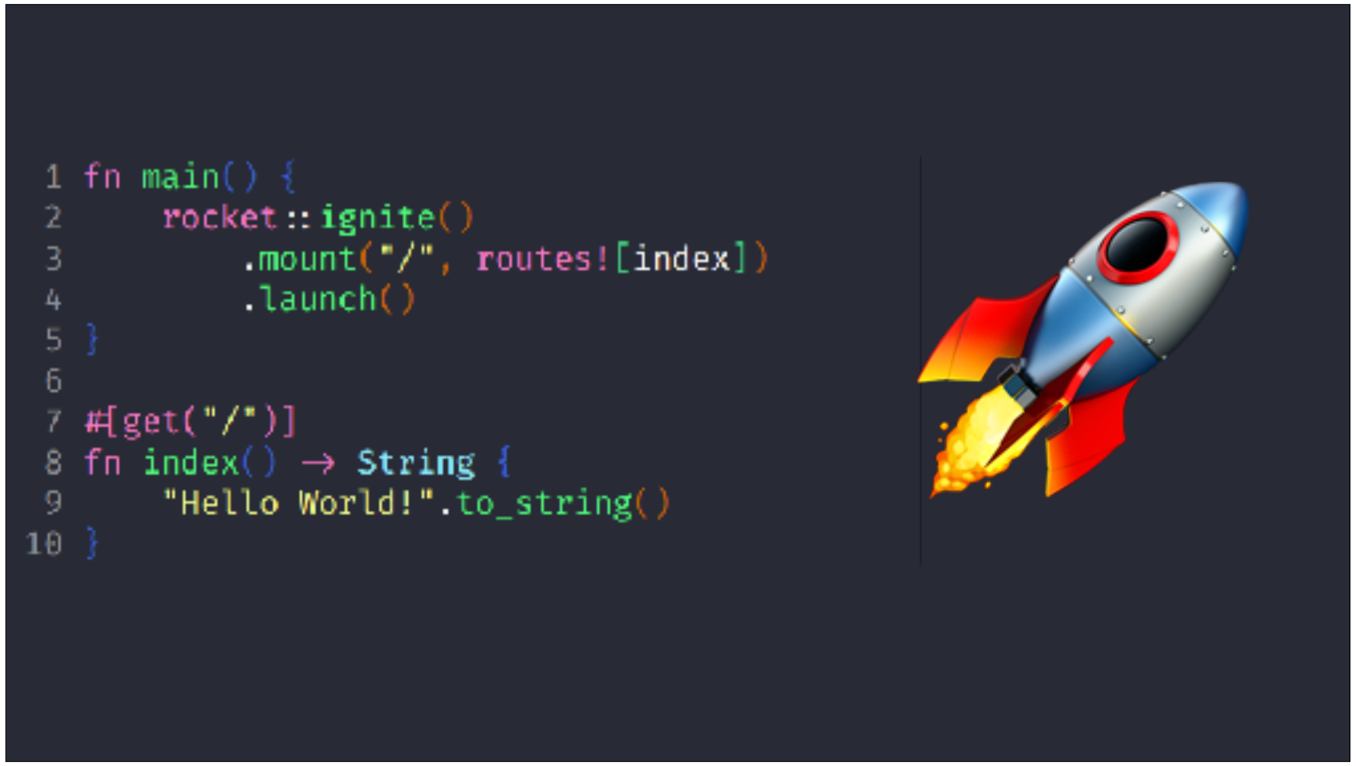
Rocket

Let’s see their main difference



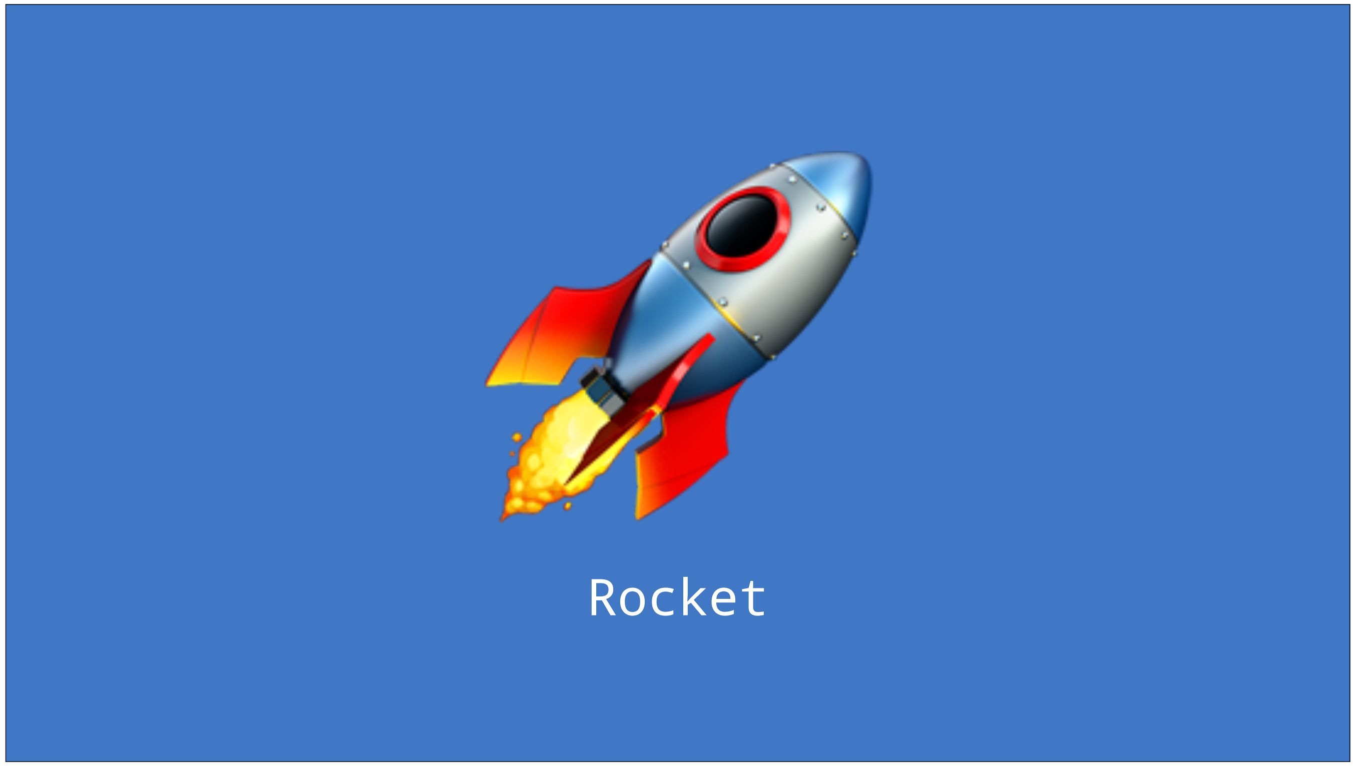
```rust
fn main() {
    // Create a router to specify which endpoint coresponds to which method
    let mut router = Router::new();
    router.get("/", index, "landing");
    //         ^^^  ^^^^^  ^^^^^^^^^
    //         path method name

    // Create a mountpoint for the application
    let mut mount = Mount::new();
    // mount the router at the root path
    mount.mount("/", router);

    // Create a request / response chain
    let mut chain = Chain::new(mount);

    let server = Iron::new(chain).http("127.0.0.1:3000");
}

fn index(request: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, None, "Hello World!")));
}
```

Iron is the older library.
It’s battle tested, sturdy, reliant, and safe. It doesn’t have a single `unsafe` block in it’s code base.
But it also adopts the age-old methodology of creating request response chains. That’s not bad per-se but it is tedious to work with.
Here is a hello world example server. A lot needs to be done to get this simple thing to run.
This is how we made web apps in rust until recently.



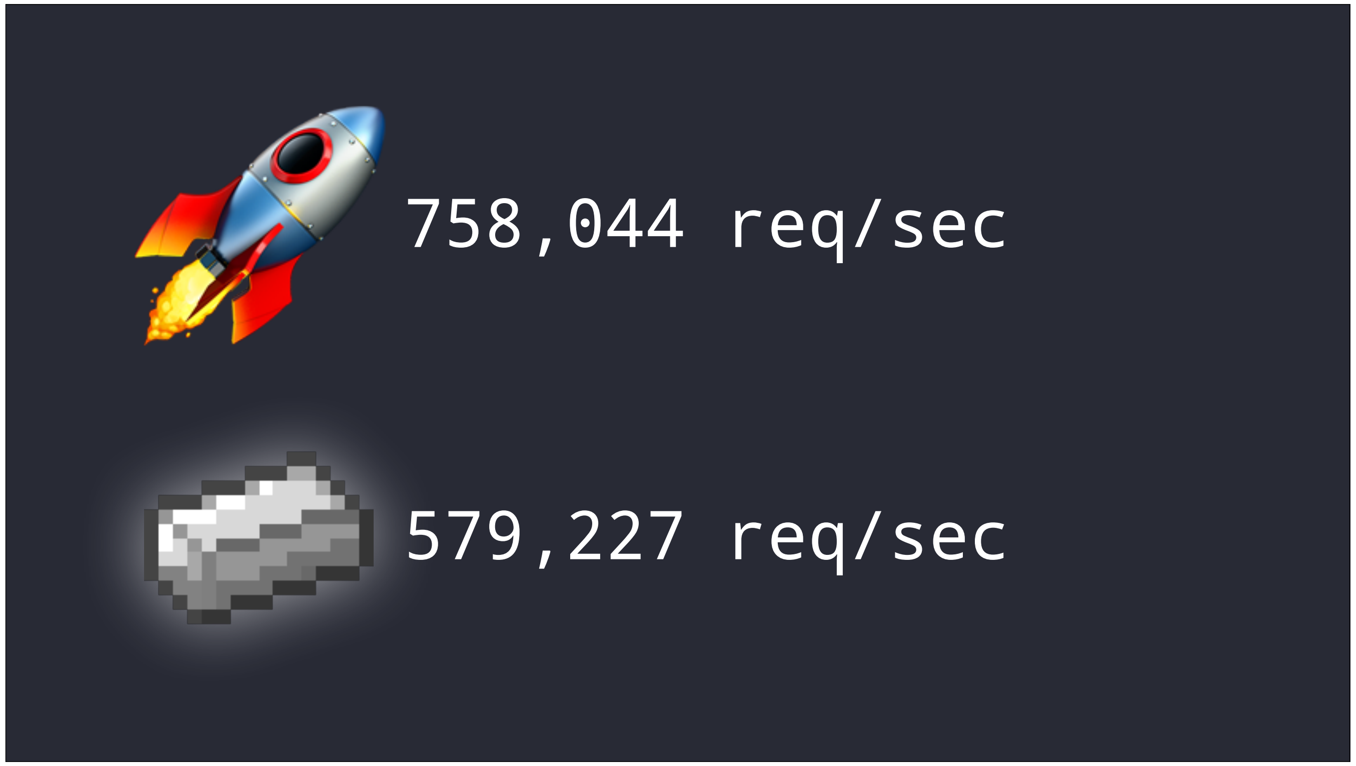
```rust
fn main() {
    rocket::ignite()
        .mount("/", routes![index])
        .launch()
}

#[get("/")]
fn index() -> String {
    "Hello World!".to_string()
}
```

The new red-hot library that is taking off. Pun intended. Is Rocket.
It adopts a filter-like architecture which I'll outline later.
This is much nicer and more readable.



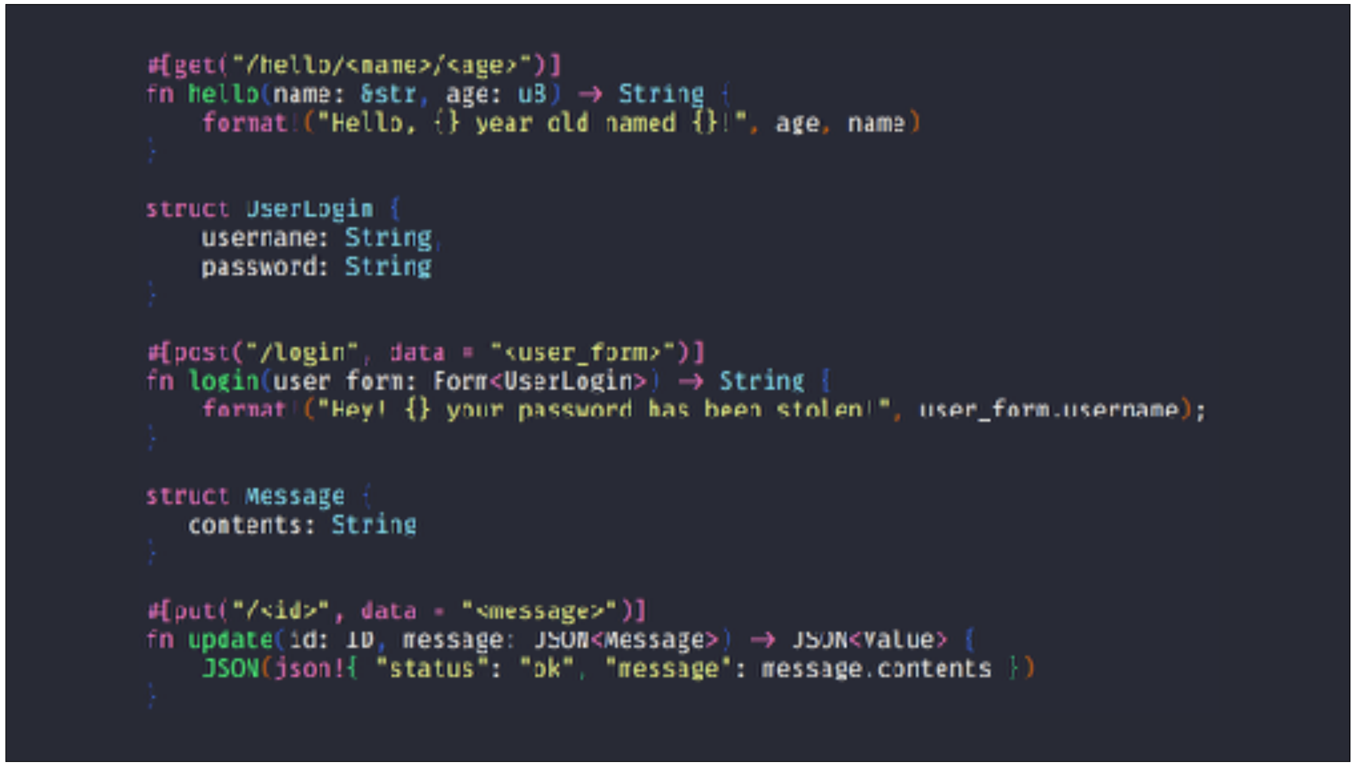
Rocket

So from this point on we will focus and explain everything using Rocket



```
758,044 req/sec

579,227 req/sec
```

Also Rocket seems to be a bit faster than Iron. Though I haven't run any benchmarks my self, I'd say that ~50% performance gain si quite significant. And yes we are already talking about ludicrous numbers here.



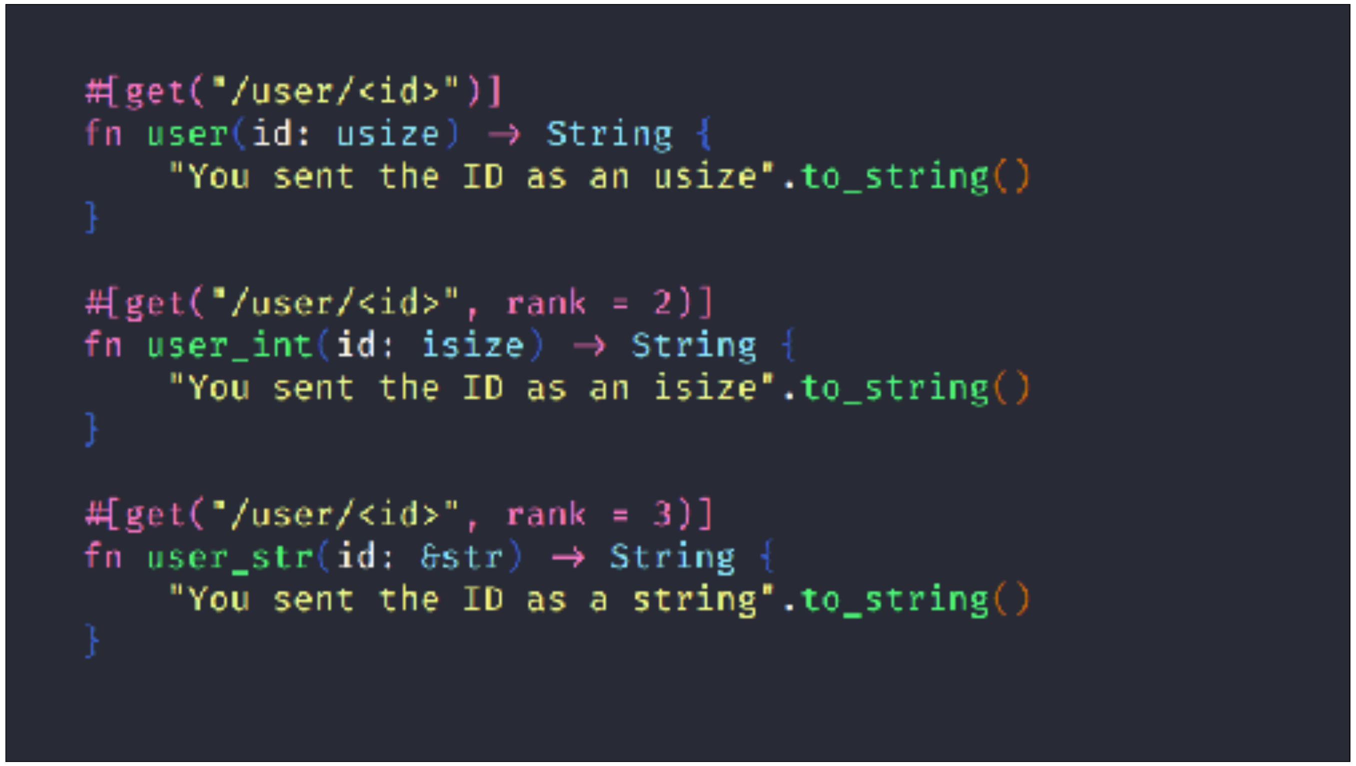
```rust
#[get("/hello/<name>/<age>")]
fn hello(name: &str, age: u8) -> String {
    format!("Hello, {} year old named {}!", age, name)
}

struct UserLogin {
    username: String,
    password: String
}

#[post("/login", data = "<user_form>")]
fn login(user form: Form<UserLogin>) -> String {
    format!("Hey! {} your password has been stolen!", user_form.username);
}

struct Message {
   contents: String
}

#[put("/<id>", data = "<message>")]
fn update(id: ID, message: JSON<Message>) -> JSON<Value> {
    JSON(json!{ "status": "ok", "message": message.contents })
}
```

Rocket provides us with tools to parser the request’s body directly. This ties in into it’s philosophy quite well.

An important note here. If UserLogin or Message can’t be parsed they will call the next defined method for that path with that interface.

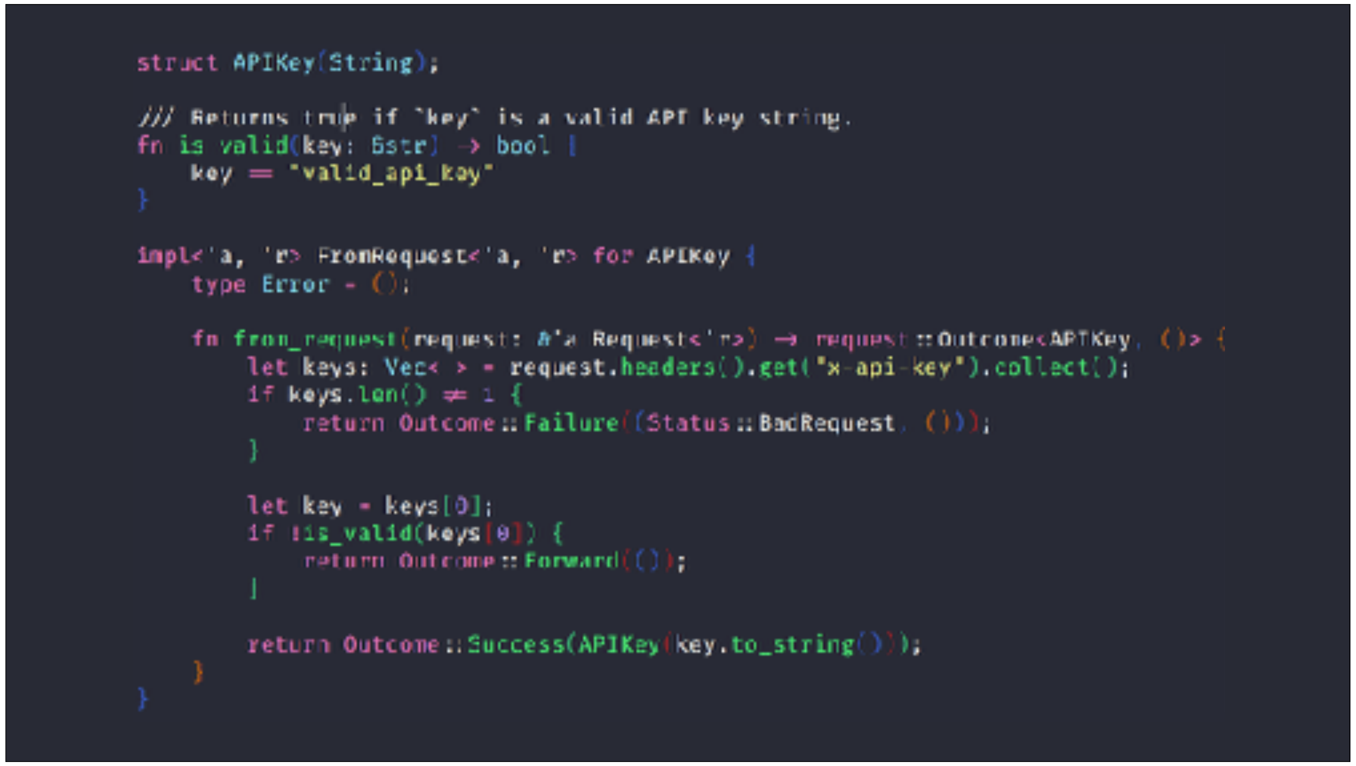
So to recap, if you send your username and password to the login method it will be executed, but if you don’t send the password it won’t get called at all. This is what the filter system is all about. Rocket calls this behaviour Request guards.

Which is fitting. Since they protect you from executing code with an unexpected input. It makes the whole process more predictable and easier to understand.



```rust
#[get("/user/<id>")]
fn user(id: usize) -> String {
    "You sent the ID as an usize".to_string()
}

#[get("/user/<id>", rank = 2)]
fn user_int(id: isize) -> String {
    "You sent the ID as an isize".to_string()
}

#[get("/user/<id>", rank = 3)]
fn user_str(id: &str) -> String {
    "You sent the ID as a string".to_string()
}
```

This is an excellent example of what I’m talking about.
The request will fall through until a corresponding method is found, or a 404 error will be returned.



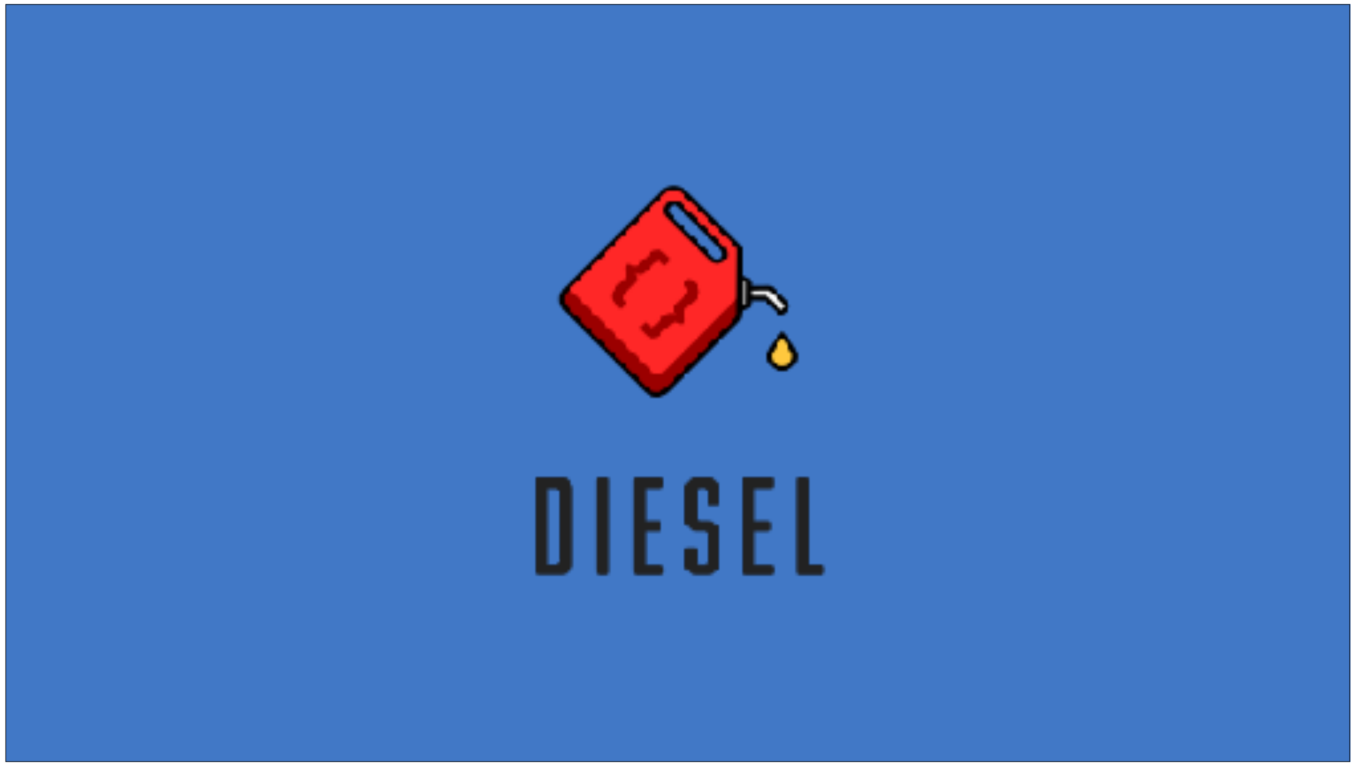
```
struct APIKey(String);

/// Returns true if `key` is a valid API key string.
fn is_valid(key: &str) -> bool {
    key == "valid_api_key"
}

impl<'a, 'r> FromRequest<'a, 'r> for APIKey {
    type Error = ();

    fn from_request(request: &'a Request<'r>) -> request::Outcome<APIKey, ()> {
        let keys: Vec<_> = request.headers().get("x-api-key").collect();
        if keys.len() != 1 {
            return Outcome::Failure((Status::BadRequest, ()));
        }

        let key = keys[0];
        if !is_valid(keys[0]) {
            return Outcome::Forward(());
        }

        return Outcome::Success(APIKey(key.to_string()));
    }
}
```

If you need to handle something complex you can do something custom.
If your struct implements the FromRequest trait then you can use it to do all the validation.

04

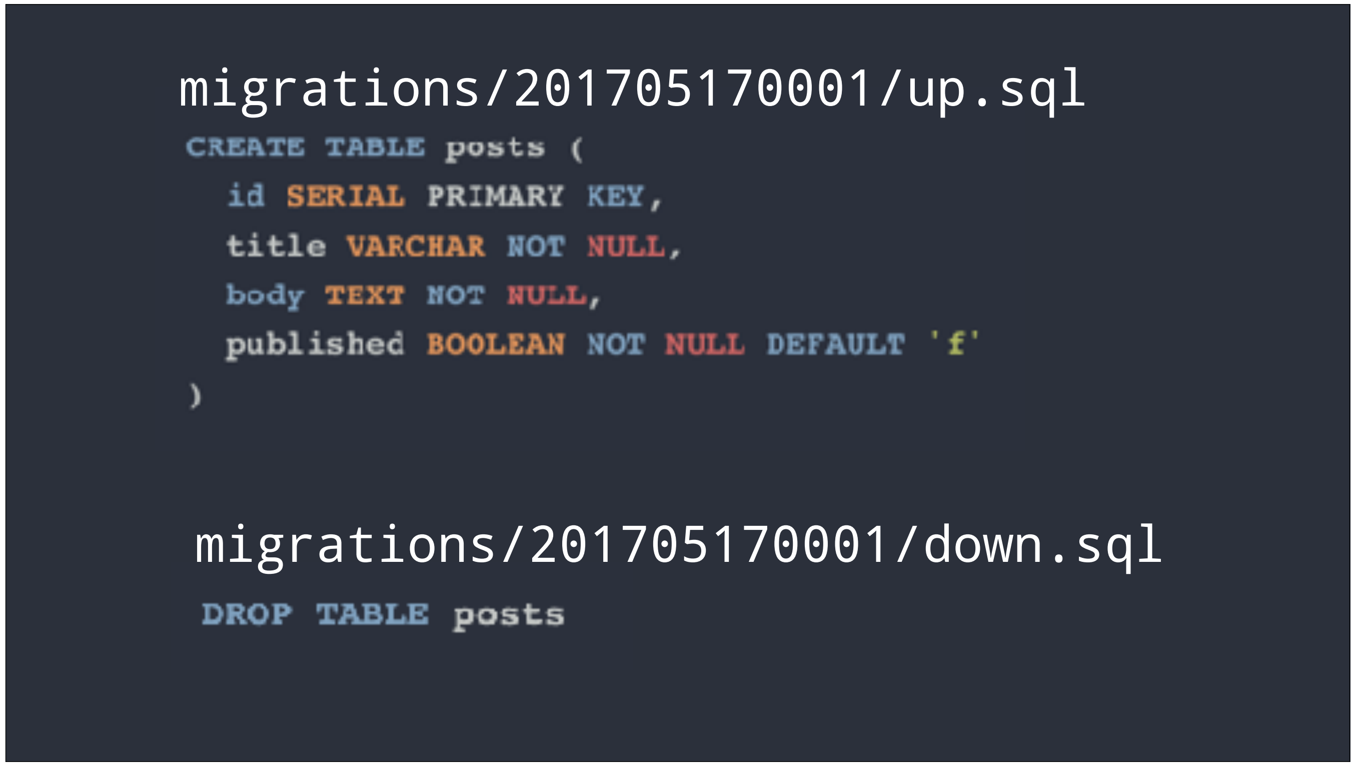
DATABASE

No we know how to make a web server, but we need to persist our information somewhere.



DIESEL

So we need to fuel our rocket with Diesel!
Diesel is an SQL runner and kind-of-ORM.



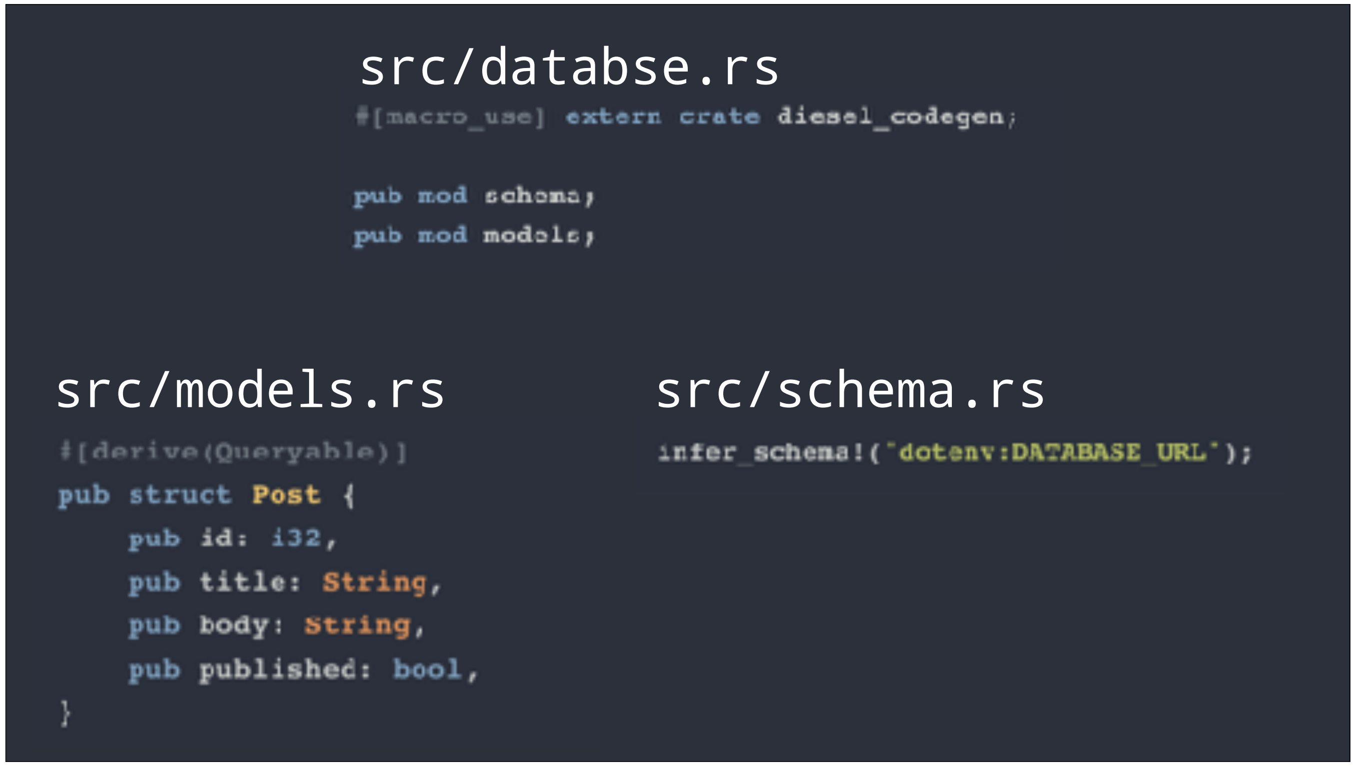
migrations/201705170001/up.sql

```sql
CREATE TABLE posts (
  id SERIAL PRIMARY KEY,
  title VARCHAR NOT NULL,
  body TEXT NOT NULL,
  published BOOLEAN NOT NULL DEFAULT 'f'
)
```

migrations/201705170001/down.sql

```sql
DROP TABLE posts
```

First thing! It handles migrations. And it does so in raw SQL.
And it provides a handy CLT tool!



src/databse.rs

```
#[macro_use] extern crate diesel_codegen;

pub mod schema;
pub mod models;
```

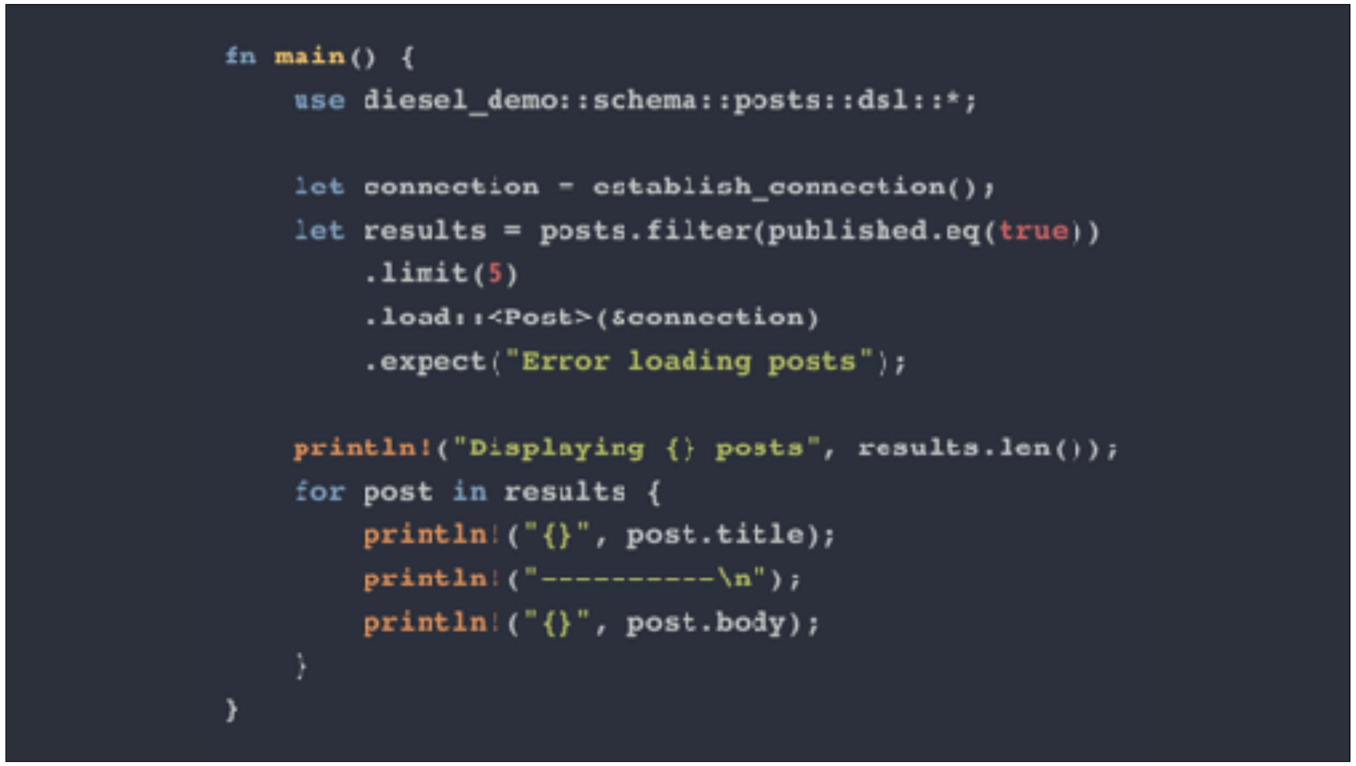
src/models.rs

```
#[derive(Queryable)]
pub struct Post {
    pub id: i32,
    pub title: String,
    pub body: String,
    pub published: bool,
}
```

src/schema.rs

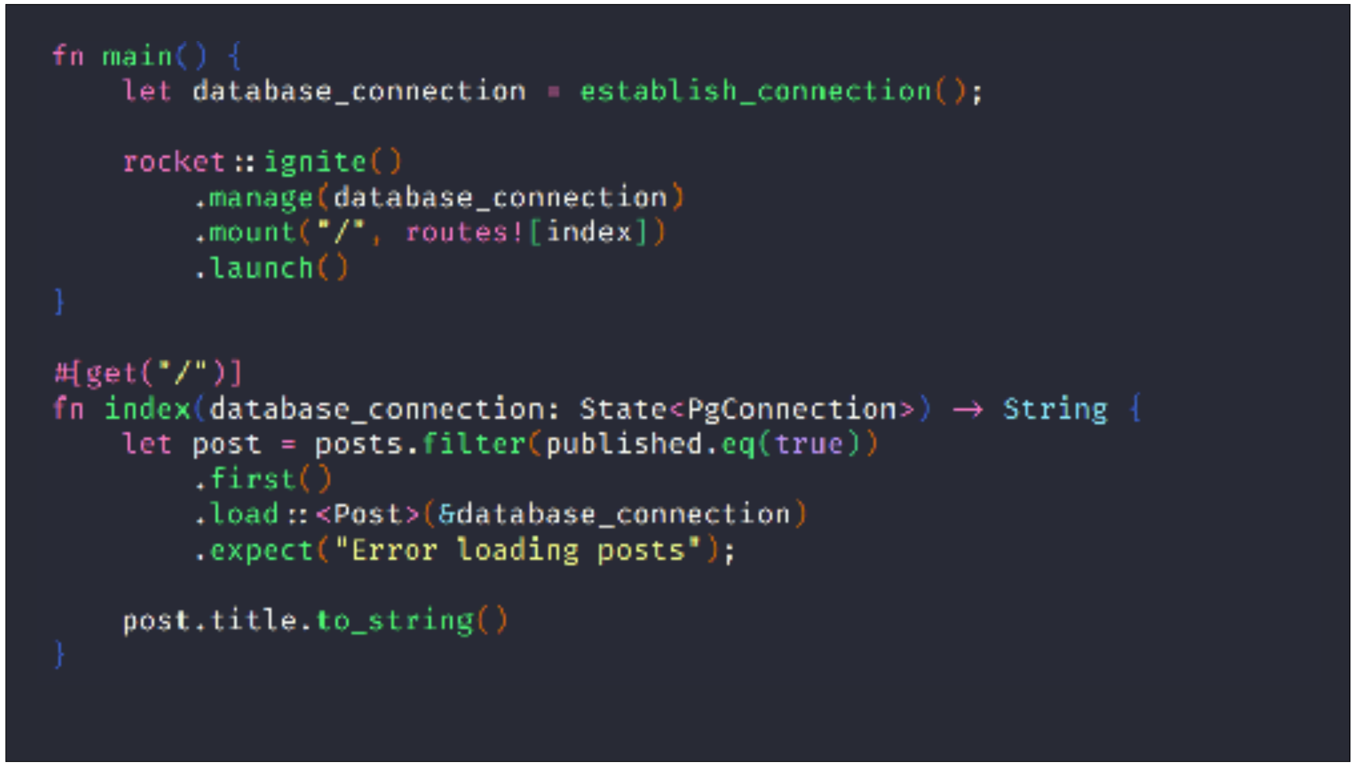
```
infer_schema!("dotenv:DATABASE_URL");
```

Now let’s define some models to use.
Please note that we require diesel_codegen and use the infere_schema macro here. This means that, at compile time, the application will attach its self to the DB anc check if the models corespond to the schema of the database. If they don’t, it will fail the build. This way we can be sure that we aren’t accesing non-existant colument, or that something is of the wrong type.



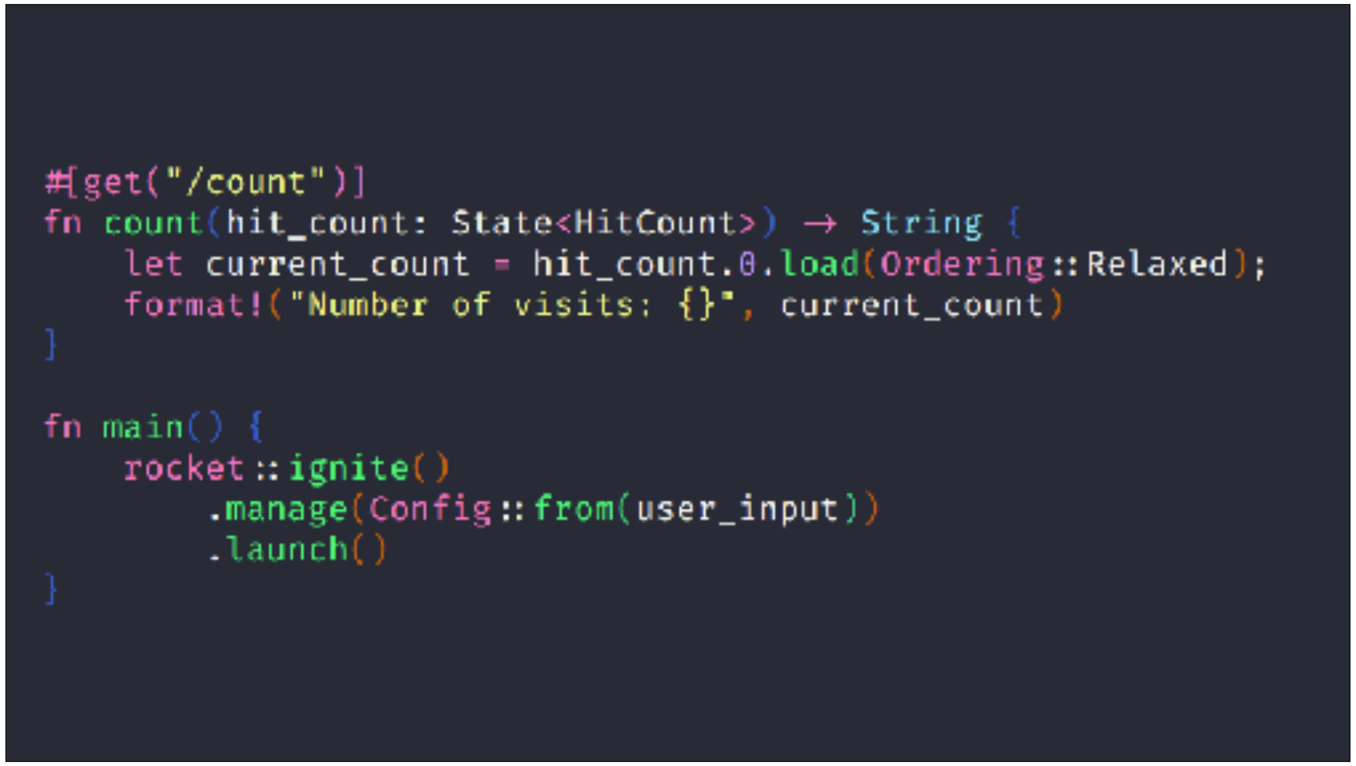
```
fn main() {
    use diesel_demo::schema::posts::dsl::*;

    let connection = establish_connection();
    let results = posts.filter(published.eq(true))
        .limit(5)
        .load::<Post>(&connection)
        .expect("Error loading posts");

    println!("Displaying {} posts", results.len());
    for post in results {
        println!("{}", post.title);
        println!("----------\n");
        println!("{}", post.body);
    }
}
```

And finally this is how you query you data!
Let’s connect this to Rocket now!



```rust
fn main() {
    let database_connection = establish_connection();

    rocket::ignite()
        .manage(database_connection)
        .mount("/", routes![index])
        .launch()
}

#[get("/")]
fn index(database_connection: State<PgConnection>) -> String {
    let post = posts.filter(published.eq(true))
        .first()
        .load::<Post>(&database_connection)
        .expect("Error loading posts");

    post.title.to_string()
}
```

We need to manage the state of the connection.
Rocket provides us with the manage method exactly for that!
With the manage method, we can create anything and pass it around to all methods. It will make it omnipresent!



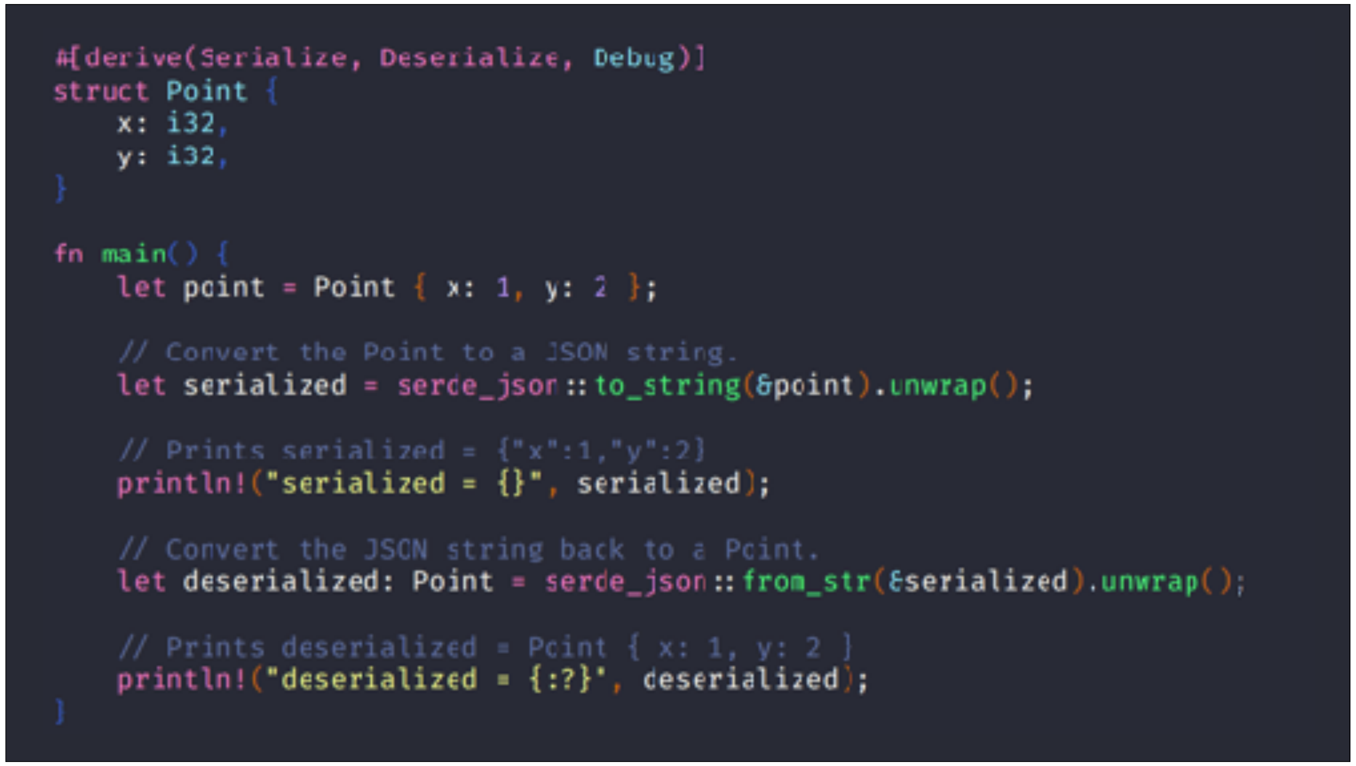
```
#[get("/count")]
fn count(hit_count: State<HitCount>) -> String {
    let current_count = hit_count.0.load(Ordering::Relaxed);
    format!("Number of visits: {}", current_count)
}

fn main() {
    rocket::ignite()
        .manage(Config::from(user_input))
        .launch()
}
```

Again, the same rule as with guards applies here. If you don’t pass the expected value to a method it won’t be executed at all.

05

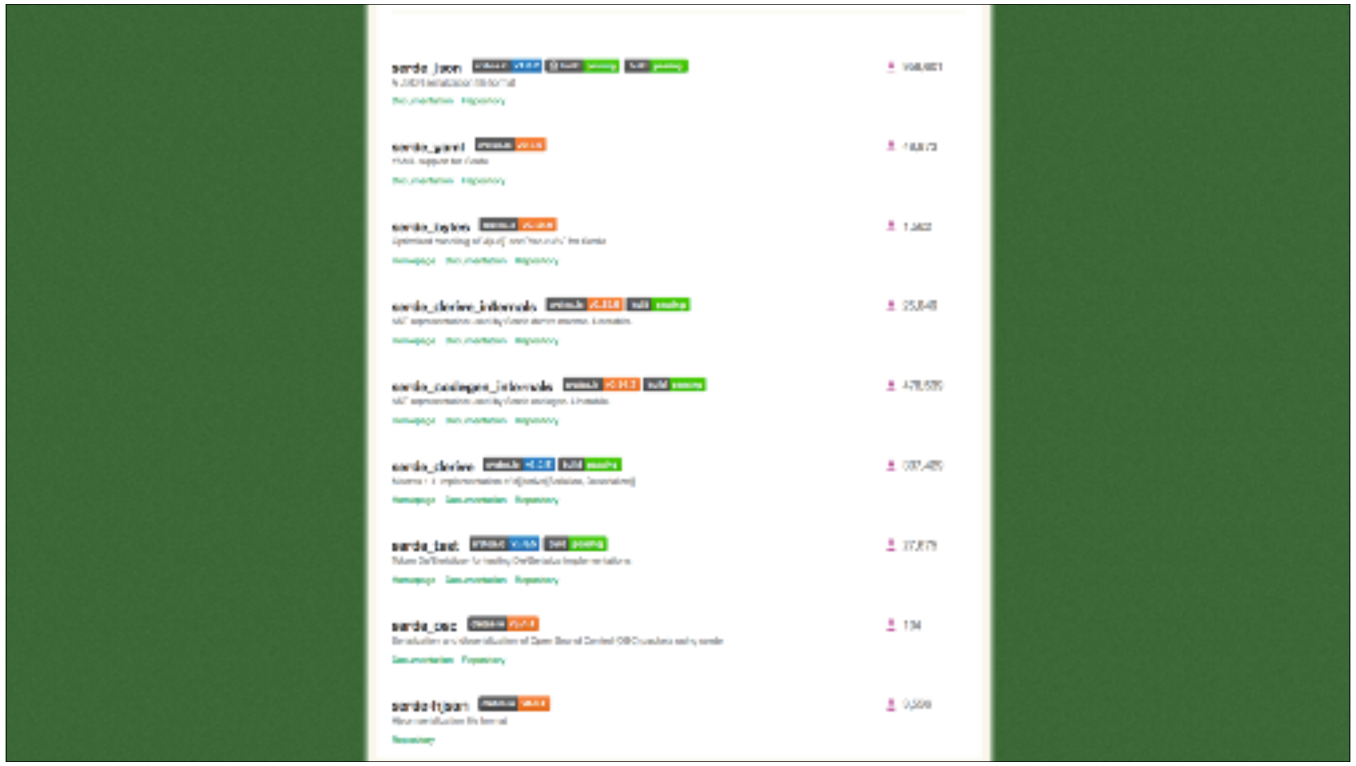
SERIALIZATION

Now we need to present the data in some way to the user.
The most popular library currently is serde.



```
#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Serde is really streight forward. I comes with a lot of incentives so that you don’t have to write code manually.



serde_json crates.io [illegible] build passing build passing 168,601
A JSON serialization file format
Documentation Repository

serde_yaml crates.io [illegible] 18,873
YAML support for Serde
Documentation Repository

serde_bytes crates.io [illegible] 1,362
Optimized handling of &[u8] and Vec<u8> for Serde
Homepage Documentation Repository

serde_derive_internals crates.io [illegible] build passing 25,848
AST representation used by Serde derive macros. Unstable.
Homepage Documentation Repository

serde_codegen_internals crates.io [illegible] build passing 478,535
AST representation used by Serde codegen. Unstable.
Homepage Documentation Repository

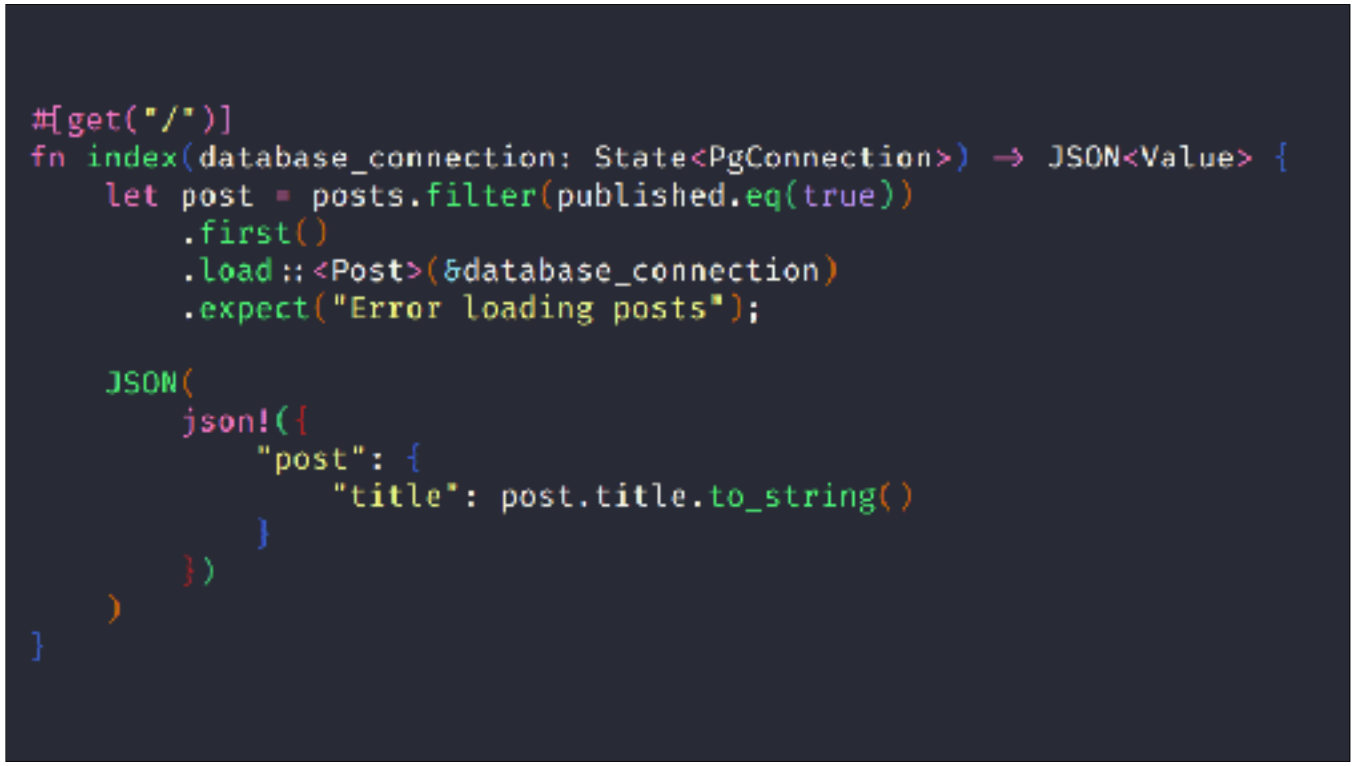
serde_derive crates.io [illegible] build passing 337,485
Macros 1.1 implementation of #[derive(Serialize, Deserialize)]
Homepage Documentation Repository

serde_test crates.io [illegible] build passing 37,879
Token De/Serializer for testing De/Serialize implementations
Homepage Documentation Repository

serde_osc crates.io [illegible] 134
Serialization and deserialization of Open Sound Control (OSC) packets using serde
Documentation Repository

serde-hjson crates.io [illegible] 9,509
Hjson serialization file format
Repository

An there is plenty of other formats supported. Ranging all the way from YAML, XML, CSV, hjson, bytes, …



```
#[get("/")]
fn index(database_connection: State<PgConnection>) -> JSON<Value> {
    let post = posts.filter(published.eq(true))
        .first()
        .load::<Post>(&database_connection)
        .expect("Error loading posts");

    JSON(
        json!({
            "post": {
                "title": post.title.to_string()
            }
        })
    )
}
```

And here it is in our Rocket app. Really streight forward.

06

ECOSYSTEM

At this point I just want to point out that there is a lot of rust libraries out there.

9,314 at the time of writing, to be exact. And I have nearly always found a library for what I was working on at the moment. Be it a library for generating JWTs, LDAP, DigitalOcean API, Coinbase API, oauth, graphql, … what not. And if there isn't any you can easily make your own with the tools I showed you.

Questions?

That would be it folks. Any questions?