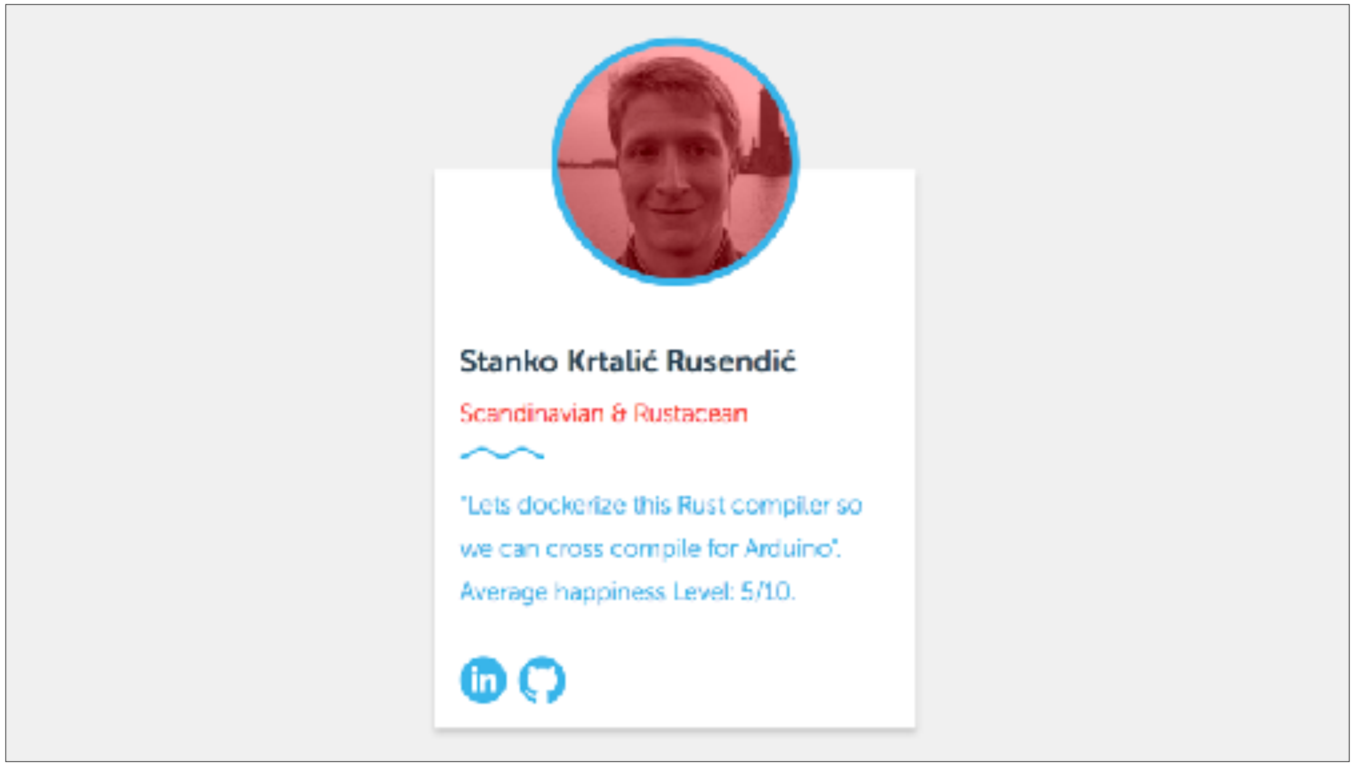
DEVELOPING A SAFER WEB WITH RUST

Hi everybody, I’ll be talking about developing a safer web with Rust.
I tend to talk quite fast, so don’t be shy and ask me questions during QA (if we get to have one), or if you prefer contact me over some kind of media.

Stanko Krtalic Rusendic

github.com/stankec
@monorkin
hi@stanko.io

My name is Stanko, I’m a web developer from Zagreb. I’ve been working at various consulting companies for the last 5 years with various technologies like Ruby, Elixir, Java, and what not.



Stanko Krtalić Rusendić

Scandinavian & Rustacean

"Lets dockerize this Rust compiler so we can cross compile for Arduino". Average happiness Level: 5/10.

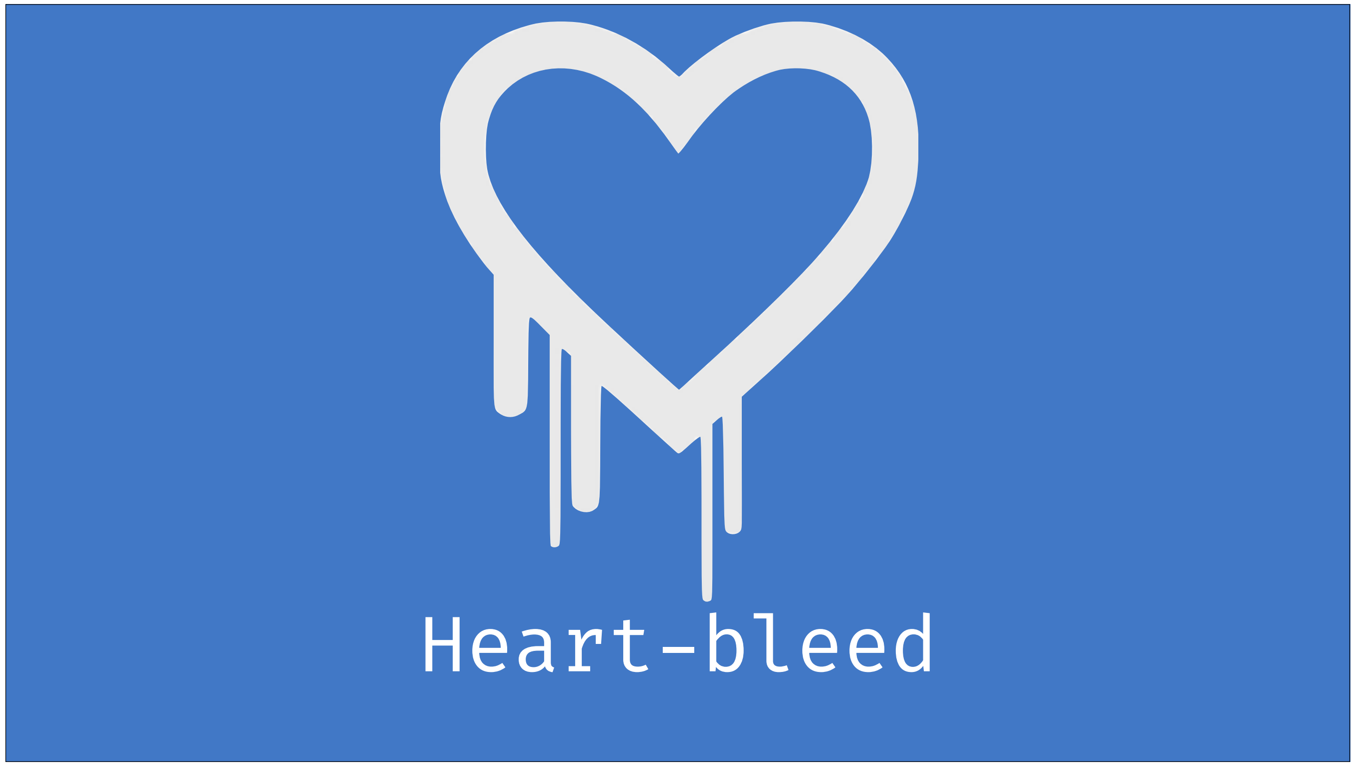
Perhaps the best description of me is this quote.

01

THE WEB OF BUGS

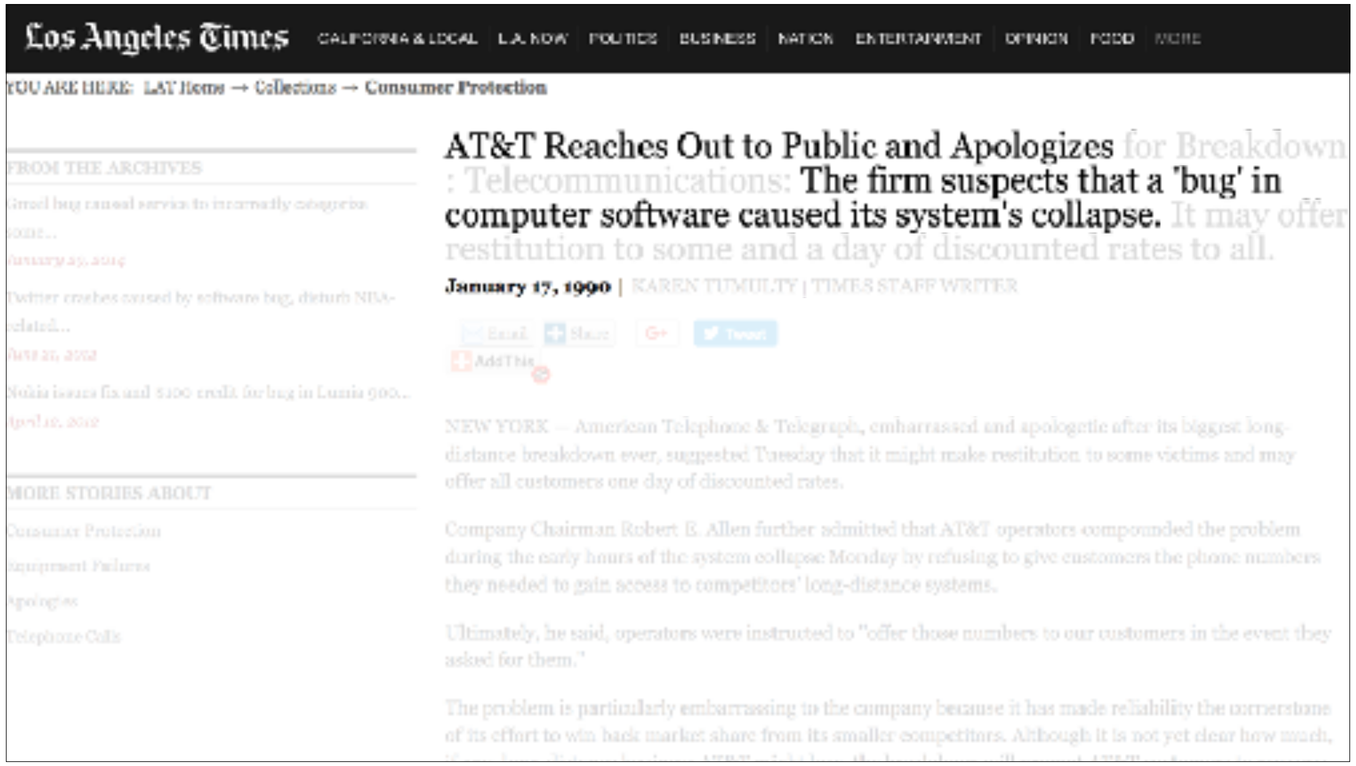
(Something only spiders love)

All web applications have bugs, but most of the bugs present in current applications could be avoided, or detected early at least.



Heart-bleed

A bug that made everybody vulnerable - Heart-bleed, was caused by a simple out-of-bounds array access. Something that could have been easily avoided and is considered to be a rookie mistake.



AT&T Reaches Out to Public and Apologizes for Breakdown : Telecommunications: The firm suspects that a 'bug' in computer software caused its system's collapse. It may offer restitution to some and a day of discounted rates to all.

January 17, 1990 | KAREN TUMULTY | TIMES STAFF WRITER

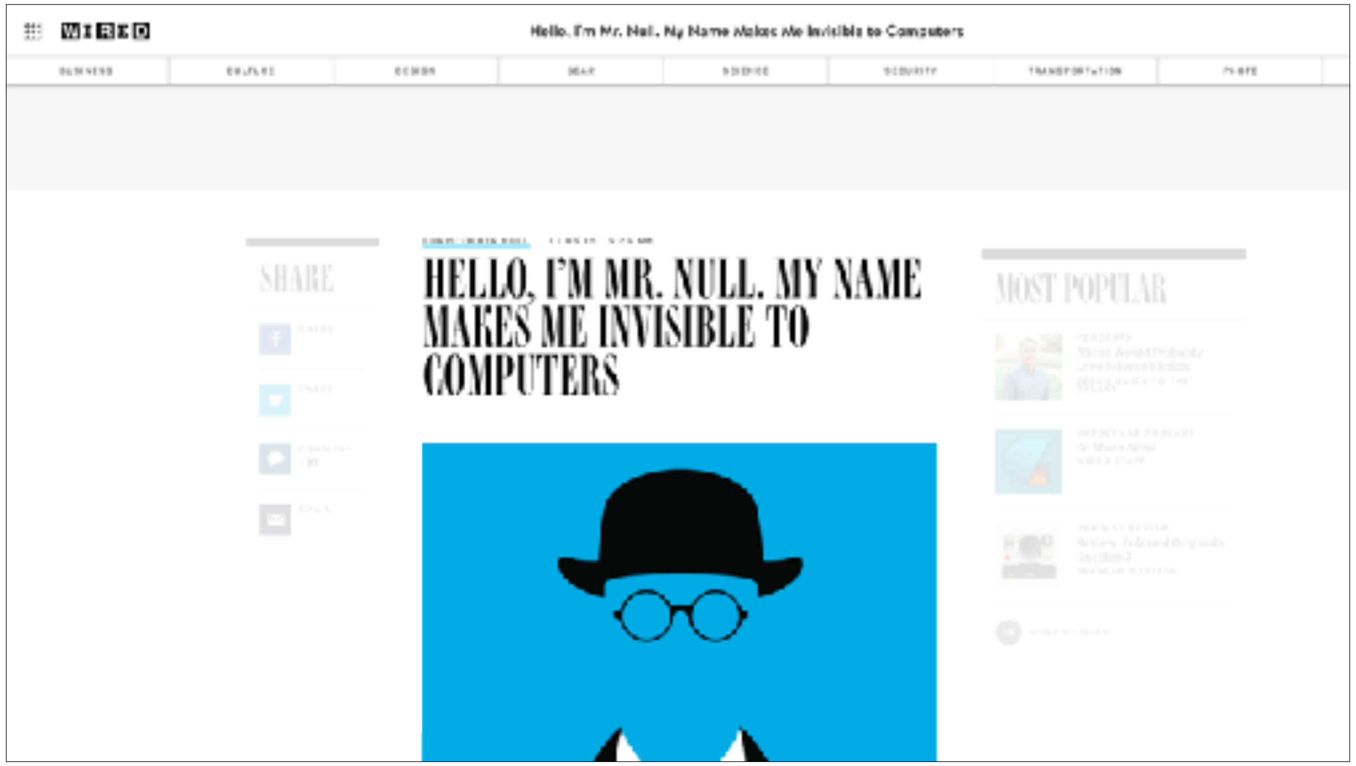
NEW YORK — American Telephone & Telegraph, embarrassed and apologetic after its biggest long-distance breakdown ever, suggested Tuesday that it might make restitution to some victims and may offer all customers one day of discounted rates.

Company Chairman Robert E. Allen further admitted that AT&T operators compounded the problem during the early hours of the system collapse Monday by refusing to give customers the phone numbers they needed to gain access to competitors' long-distance systems.

Ultimately, he said, operators were instructed to "offer those numbers to our customers in the event they asked for them."

The problem is particularly embarrassing to the company because it has made reliability the cornerstone of its effort to win back market share from its smaller competitors. Although it is not yet clear how much,

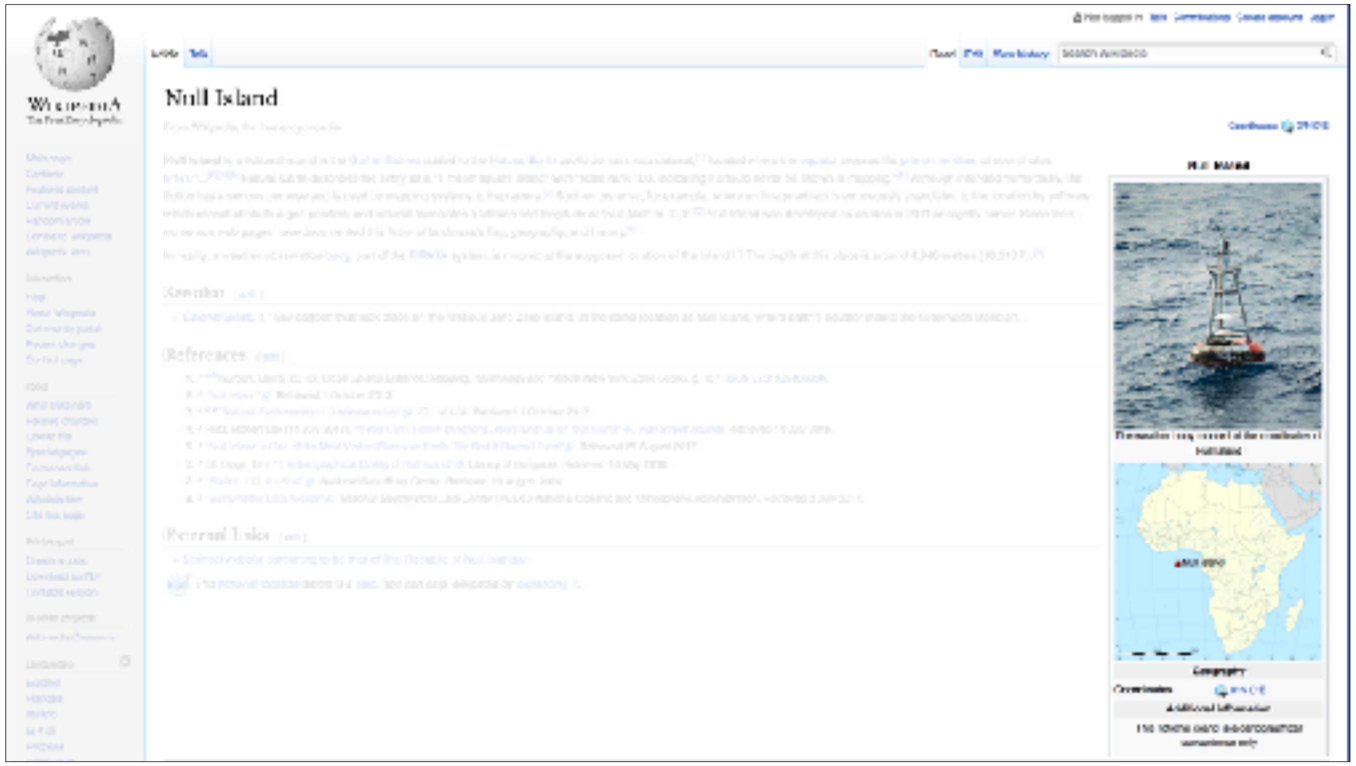
In the 90’s AT&T’s system collapsed due to a bug that was essentially a data race condition in it’s server cluster



Hello, I'm Mr. Null. My Name Makes Me Invisible to Computers

HELLO, I'M MR. NULL. MY NAME MAKES ME INVISIBLE TO COMPUTERS

And then there is the NULL. It causes so many hilarious bugs to this day that I would argue that it's the most prominent bug ever. For example, this man can't book a flight, ever. Because most airline systems think that his surname is a NULL value.

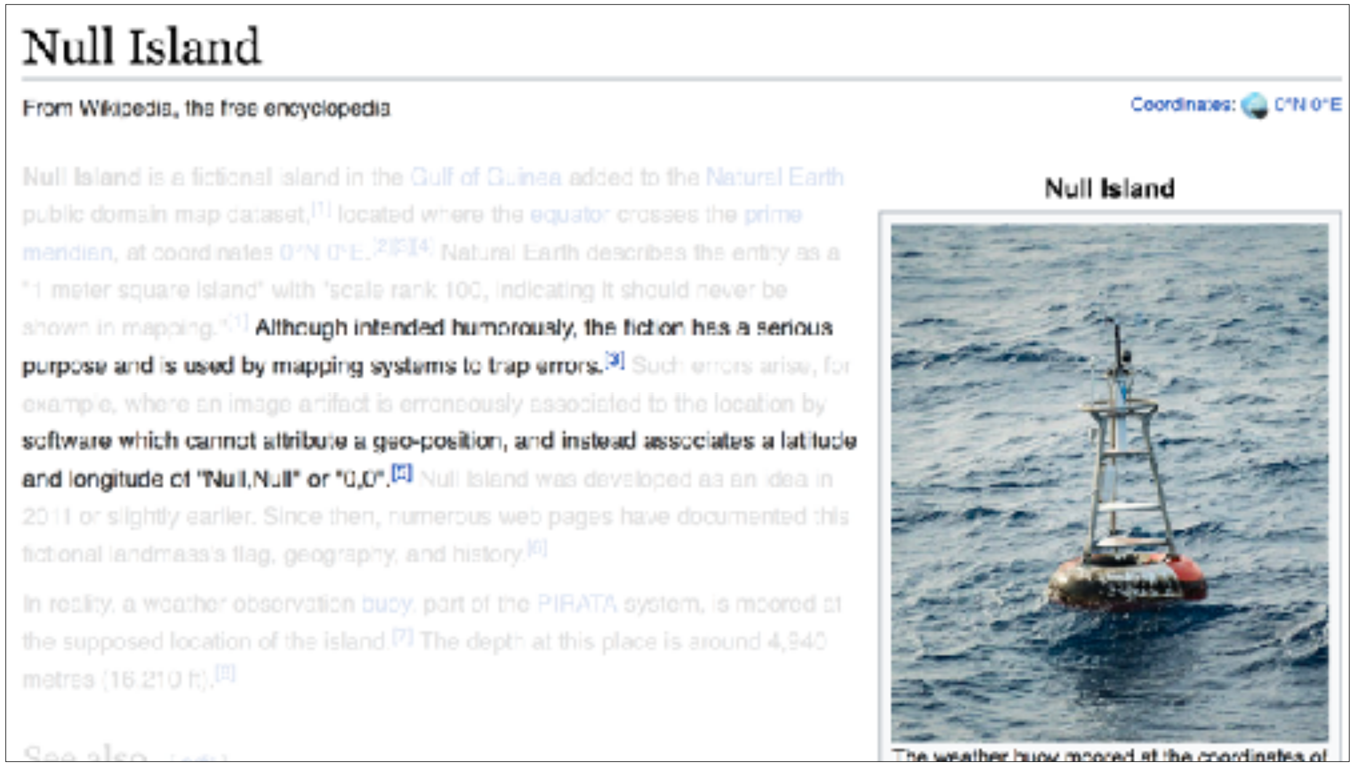


Null Island

References

External links

This problem is so prevalent that there exists such a thing as NULL island. It’s a buoy in the middle of the ocean. Located at 0 LAT and 0 LNG.



Null Island

From Wikipedia, the free encyclopedia

Coordinates: 0°N 0°E

Null Island is a fictional island in the Gulf of Guinea added to the Natural Earth public domain map dataset,[1] located where the equator crosses the prime meridian, at coordinates 0°N 0°E.[2][3][4] Natural Earth describes the entity as a "1 meter square island" with "scale rank 100, indicating it should never be shown in mapping."[1] Although intended humorously, the fiction has a serious purpose and is used by mapping systems to trap errors.[3] Such errors arise, for example, where an image artifact is erroneously associated to the location by software which cannot attribute a geo-position, and instead associates a latitude and longitude of "Null,Null" or "0,0".[5] Null Island was developed as an idea in 2011 or slightly earlier. Since then, numerous web pages have documented this fictional landmass's flag, geography, and history.[6]

In reality, a weather observation buoy, part of the PIRATA system, is moored at the supposed location of the island.[7] The depth at this place is around 4,940 metres (16,210 ft).[8]

See also

Null Island

The weather buoy moored at the coordinates of

NULL island started out as a joke in 2011., but is now a real world thing. It's used to trap NULL errors in positional systems, e.g. when your GPS can't acquire a signal and presumes you are at NULL, NULL or 0,0.

Technology

UK hacker exploits online bank loophole to steal £100,000

A UK hacker has been jailed for stealing almost £100,000 from a bank by exploiting a bug in the bank's online banking system.

[illegible]

Top Stories

Police hunt for Las Vegas killer's motive

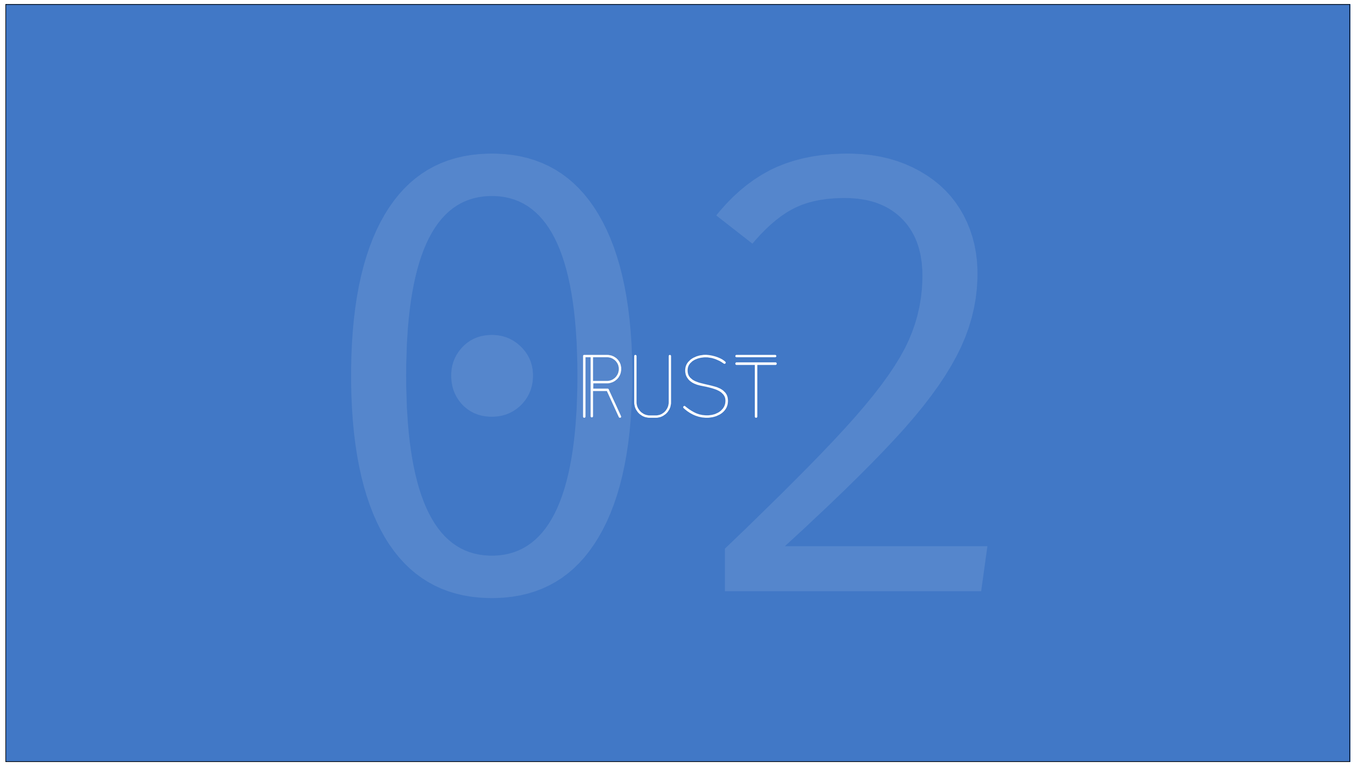
US musician Tom Petty dies aged 66

Anti-police anger grips Catalonia

Features

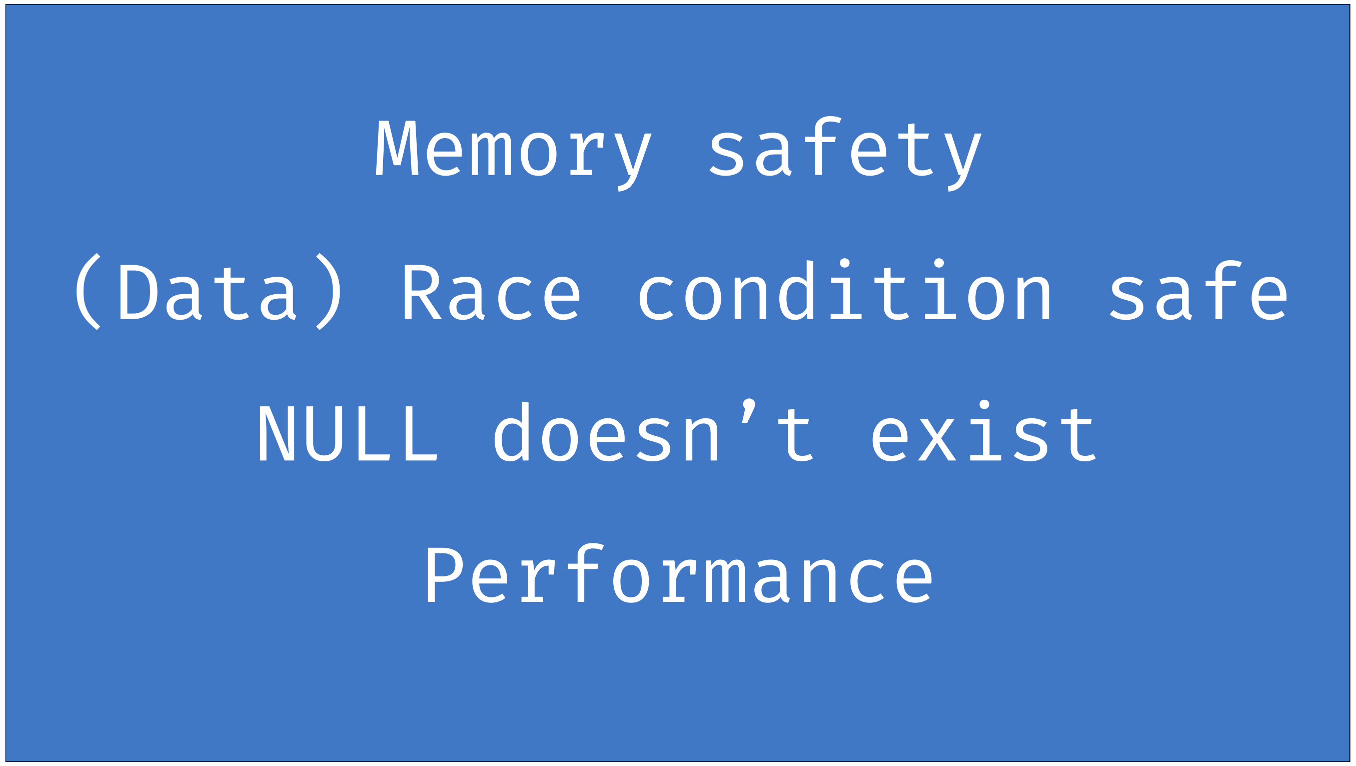
Las Vegas shooting - what we know

With all these bugs in systems that we think of as essential and battle tested how can we presume to build any kind of service that somebody won’t break and exploit. Take this online banking exploit, where a hacker managed to create money out of thin air. How can you be sure that your current project doesn’t have an equivalent bug?



02 • RUST

To give a shot answer to that question. You can't. But you can try to prevent those bugs. Which brings us to Rust. I discovered Rust back in 2015., it still wasn't stable at the time, but it promised some things that I though were impossible at the time.



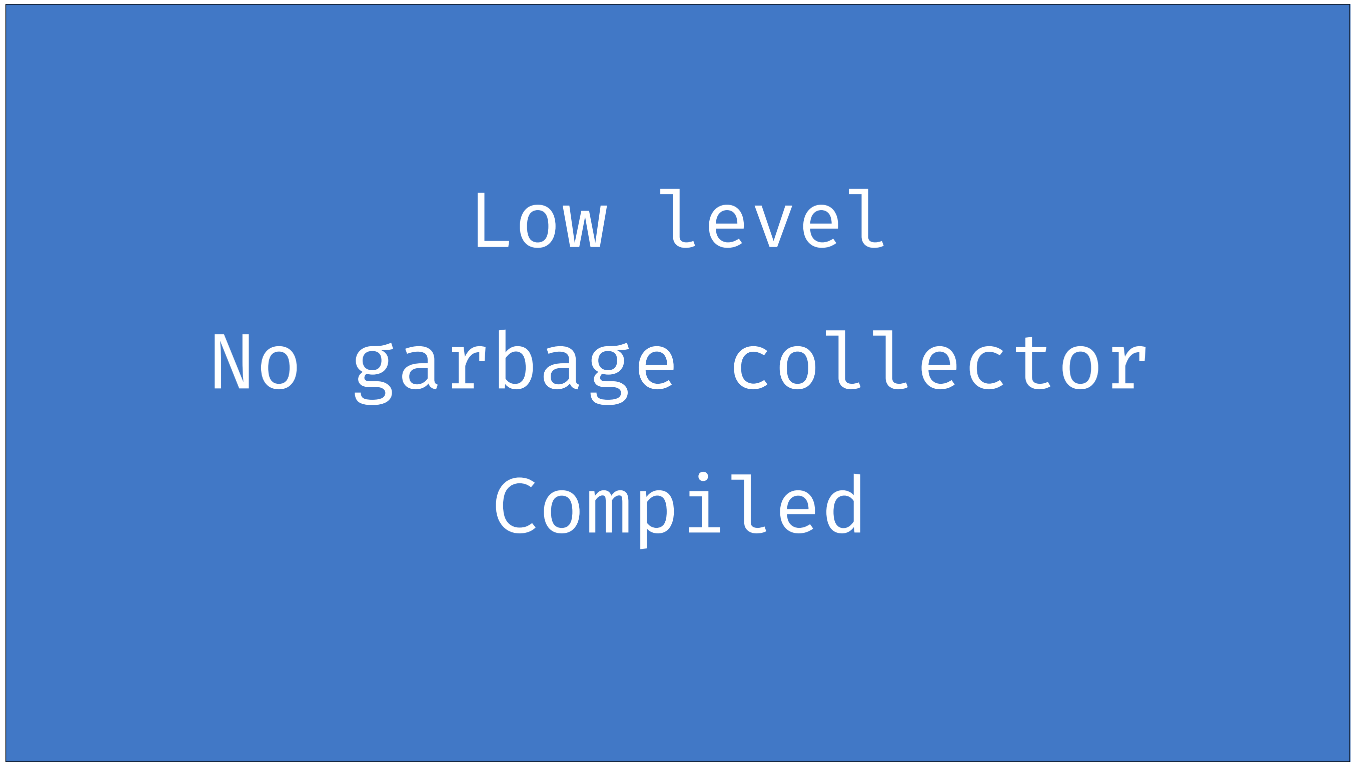
Memory safety

(Data) Race condition safe

NULL doesn’t exist

Performance

It promised memory safety and the absence of data races with no performance penalty. Just think about it for a second. If SSL would have been written in Rust, HeartBleed would have never been a thing. AT&T’s server cluster would have never failed in such a catastrophic manner. That guy could actually book a flight! And everybody would be sad to see NULL island is gone.

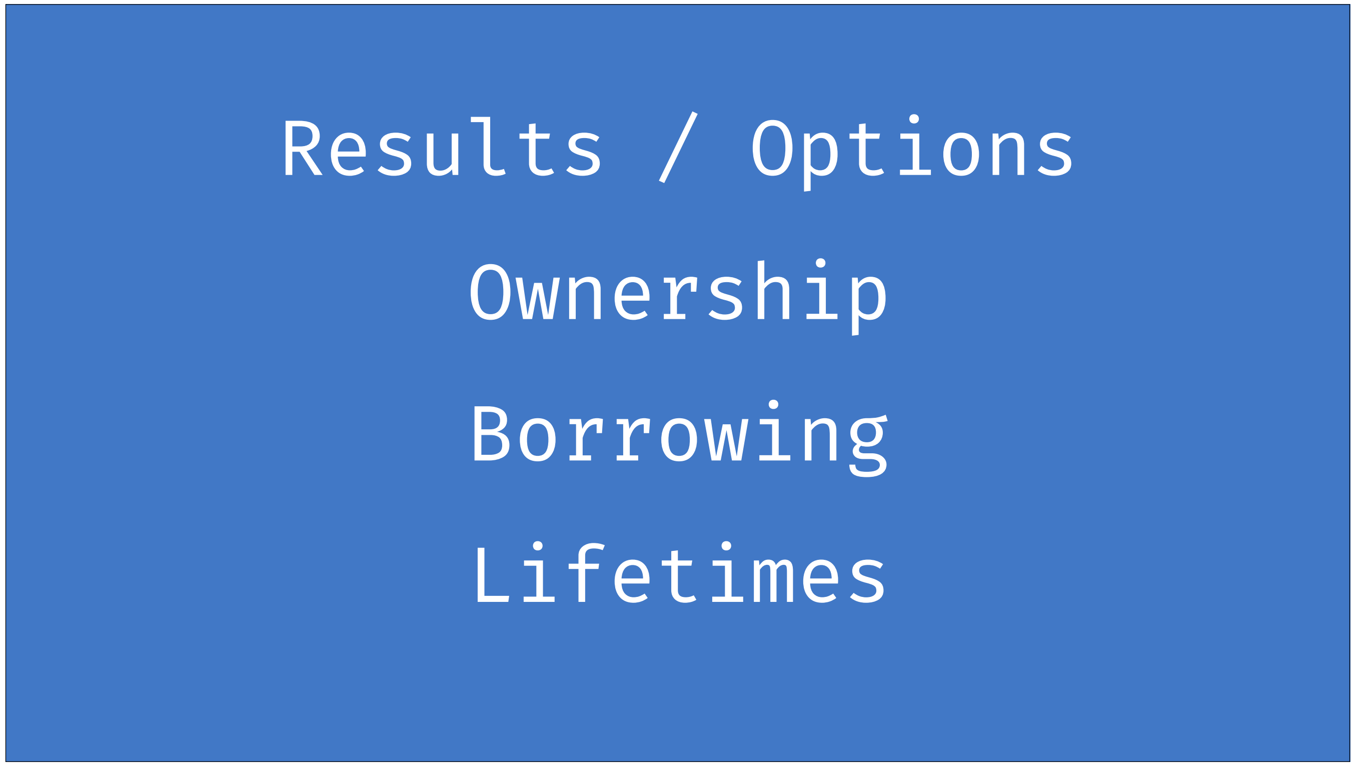


Low level

No garbage collector

Compiled

Those guarantees come at a price, the language is low level and it’s compiled. Which a few years ago would have been a no-go for a new project, but with the rise go Google’s Go language it’s becoming more acceptable lately. And these, as of yesterday’s, weaknesses are starting to become strengths when it comes to web development. Also, note that the absence of a garbage collector doesn’t mean that you have to manage memory by hand. You can just create structs and Rust will do it’s magic and deterministically remove it from memory when it’s not used any more.



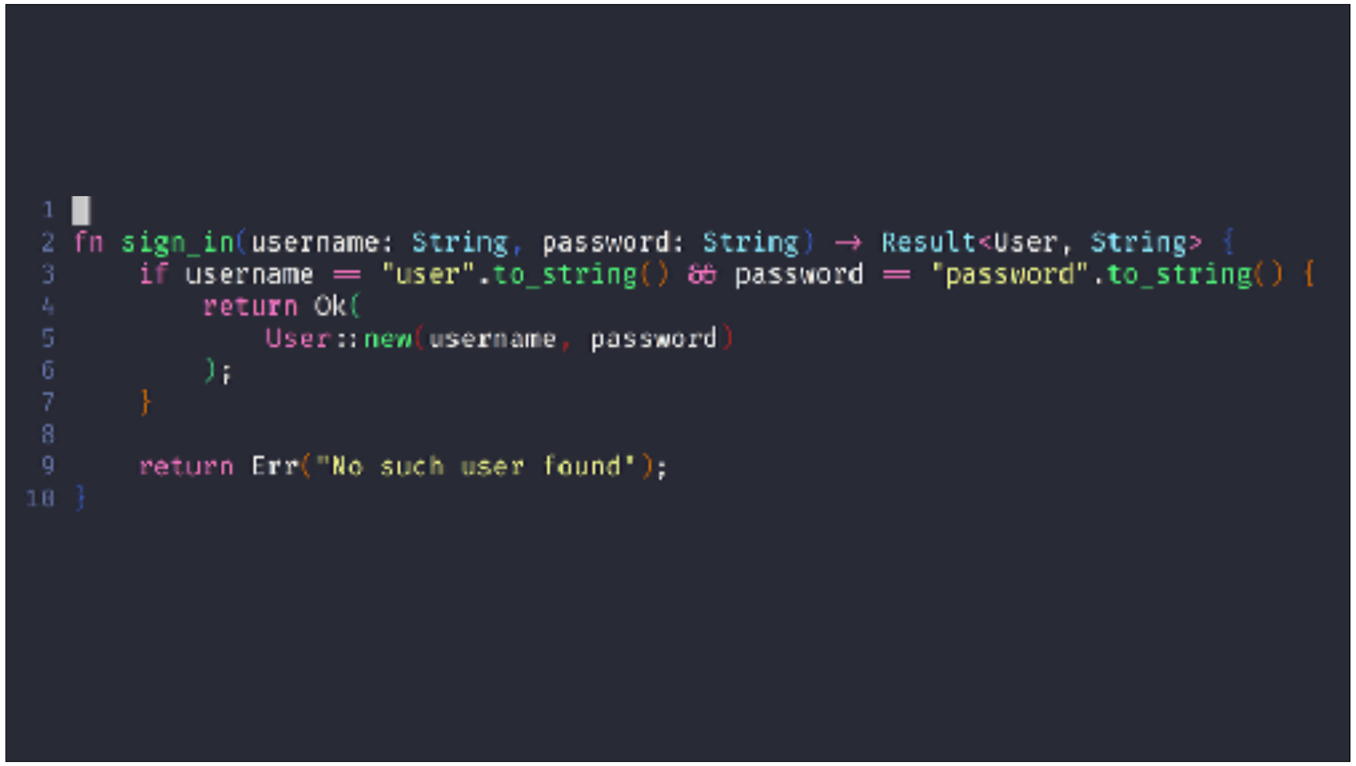
Results / Options

Ownership

Borrowing

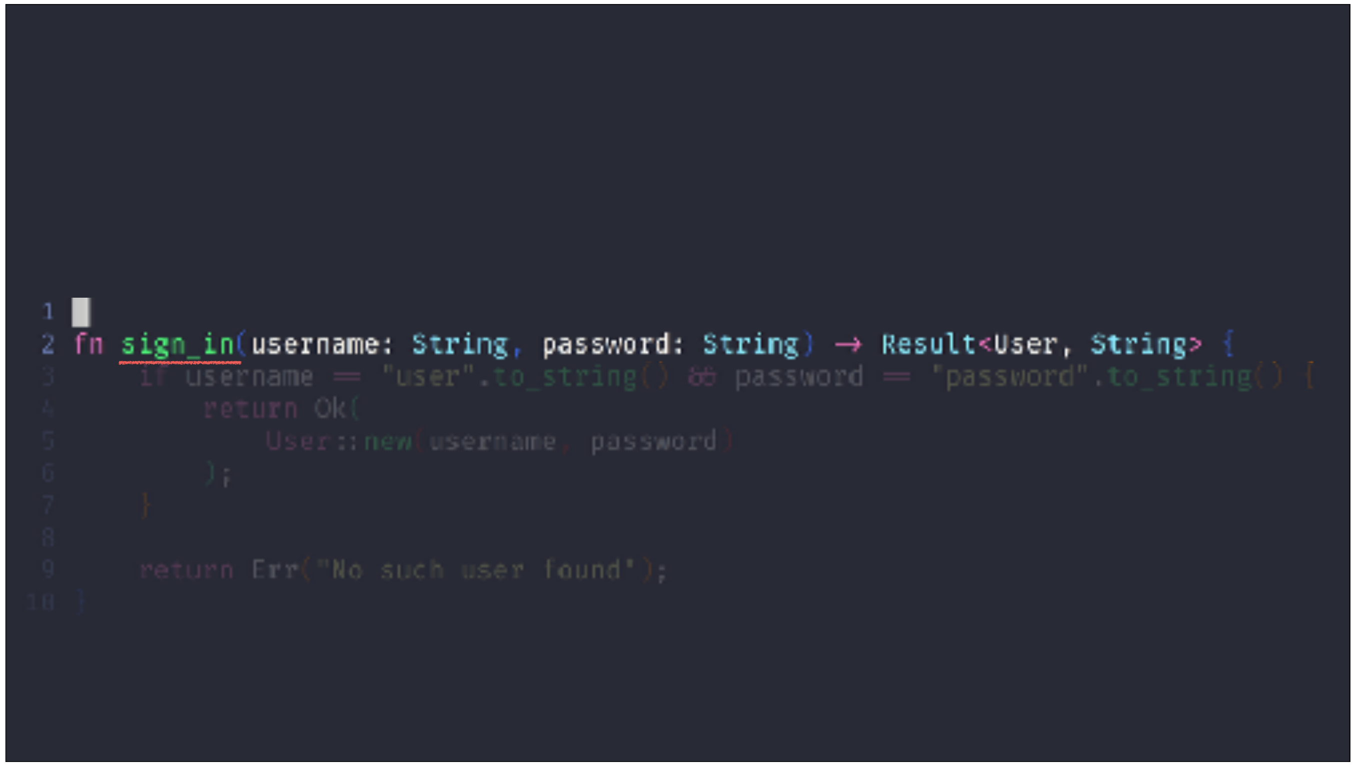
Lifetimes

This kind of magic is accomplished with the following concepts.
Results and options replace the NULL value. They are comparable to Haskell’s Maybe.
Ownership and borrowing, which are uniq to Rust, solve data race conditions and make threading more sane.
While Lifetimes abolish the need for a garbage collector.



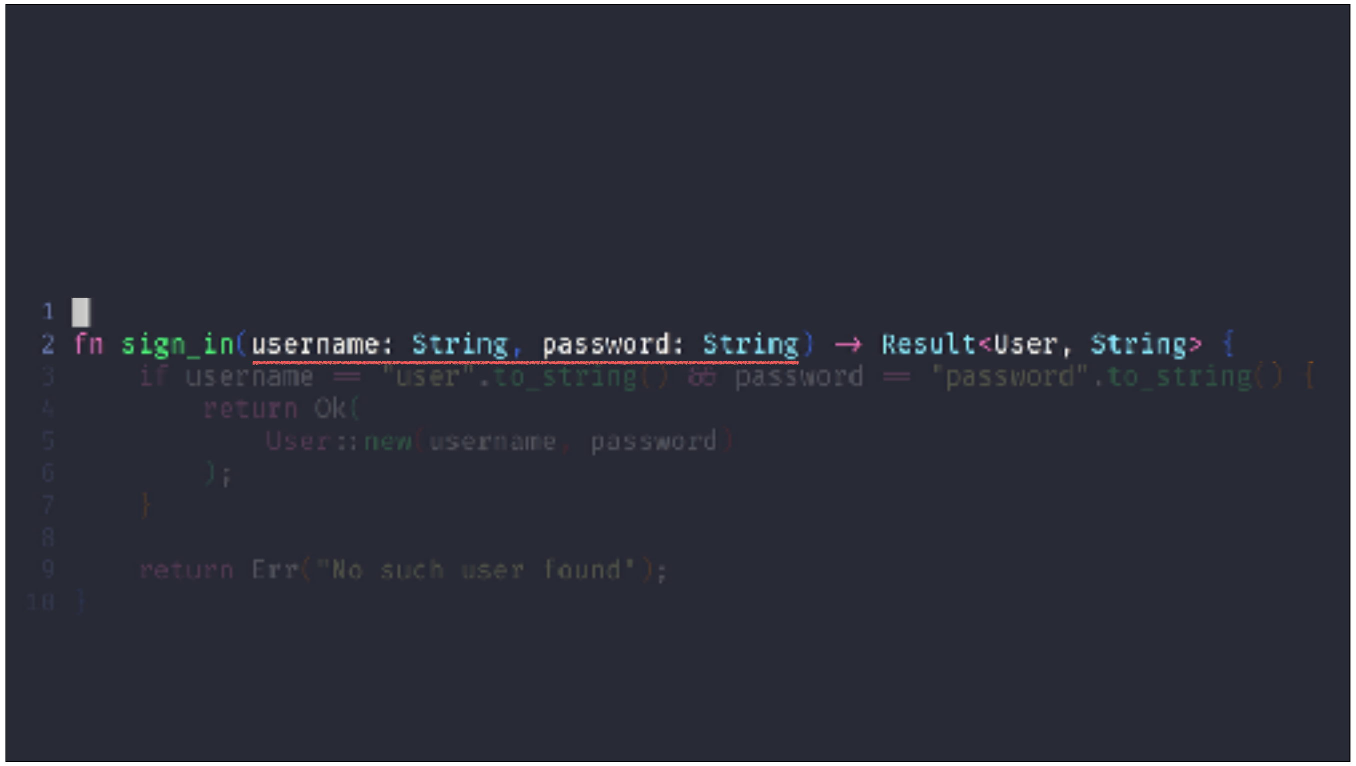
```
fn sign_in(username: String, password: String) -> Result<User, String> {
    if username == "user".to_string() && password == "password".to_string() {
        return Ok(
            User::new(username, password)
        );
    }

    return Err("No such user found");
}
```

Here is an example of a function returning a result



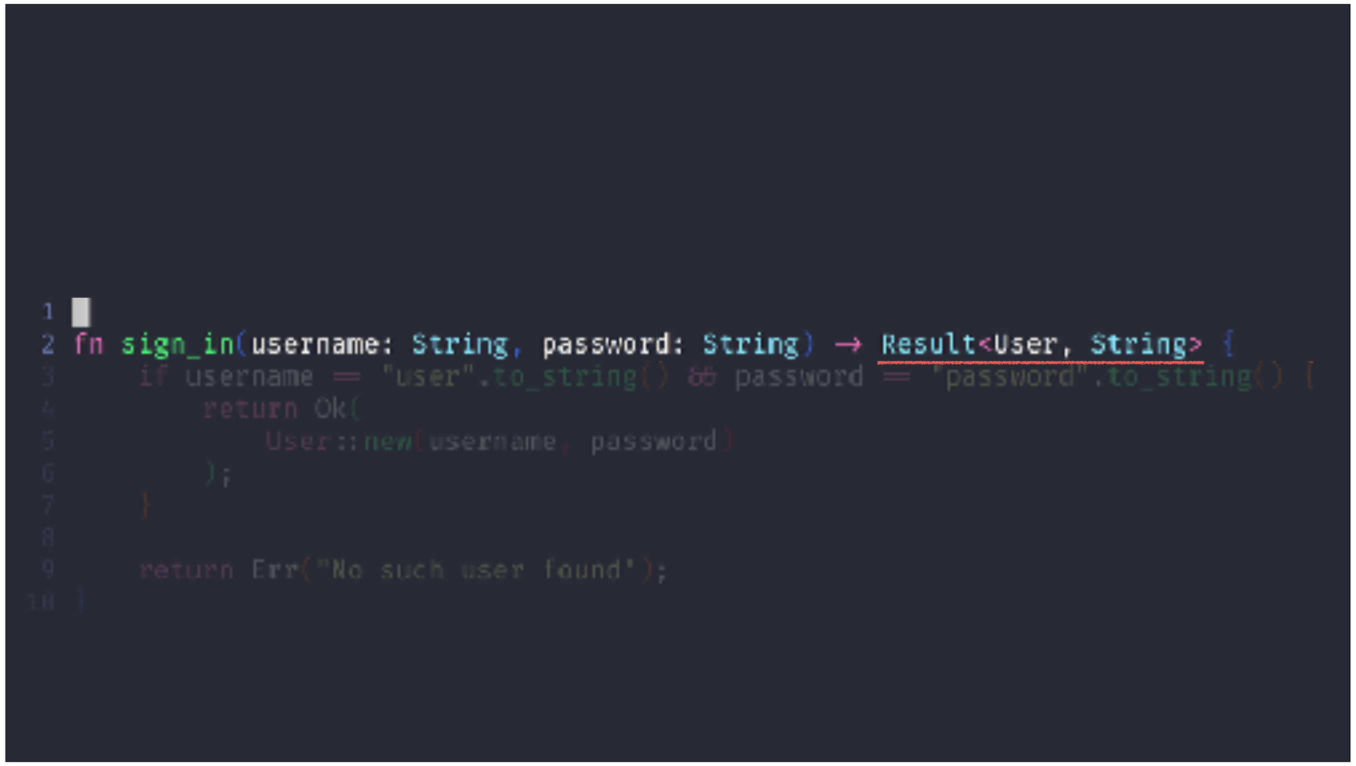
```rust

fn sign_in(username: String, password: String) -> Result<User, String> {
    if username == "user".to_string() && password == "password".to_string() {
        return Ok(
            User::new(username, password)
        );
    }

    return Err("No such user found");
}
```

The name of the function is sign_in



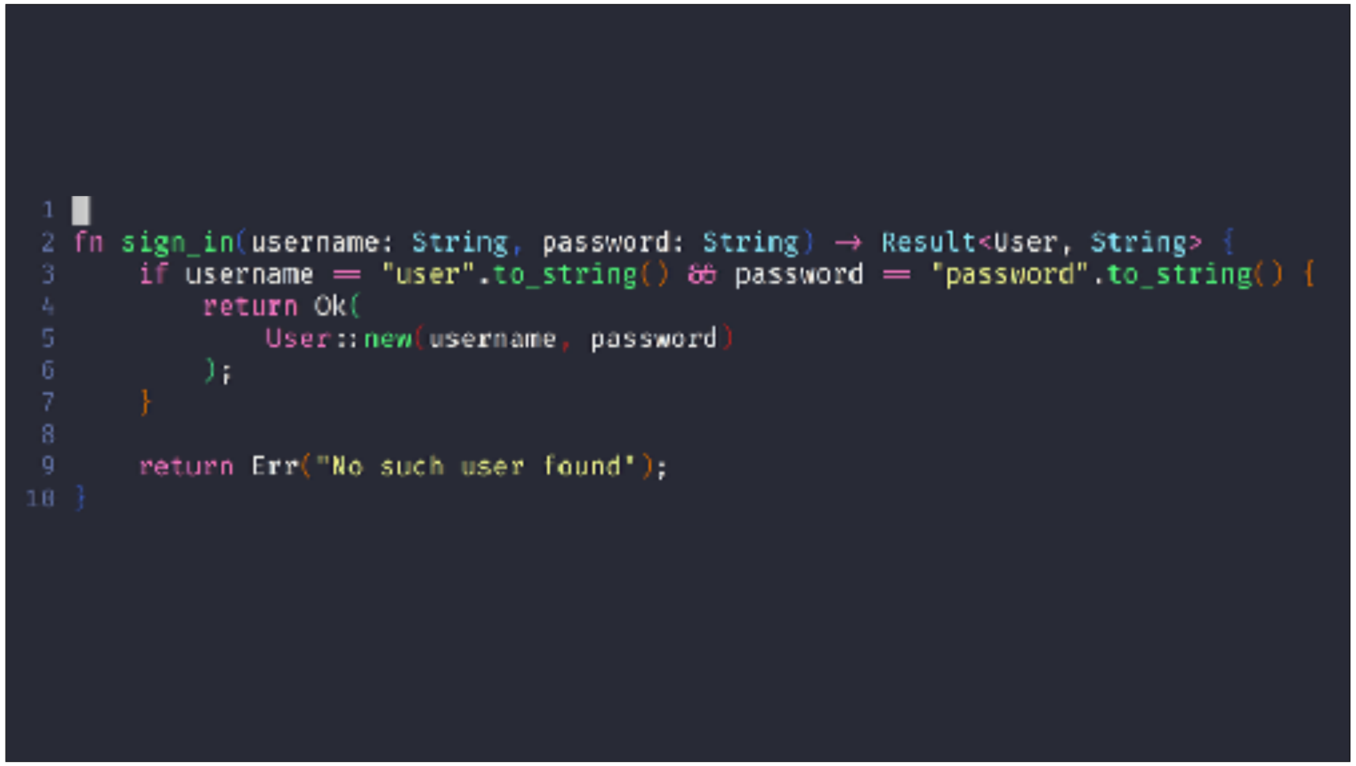
```

fn sign_in(username: String, password: String) → Result<User, String> {
    if username == "user".to_string() && password == "password".to_string() {
        return Ok(
            User::new(username, password)
        );
    }

    return Err("No such user found");
}
```

It accepts two arguments of type String called username and password



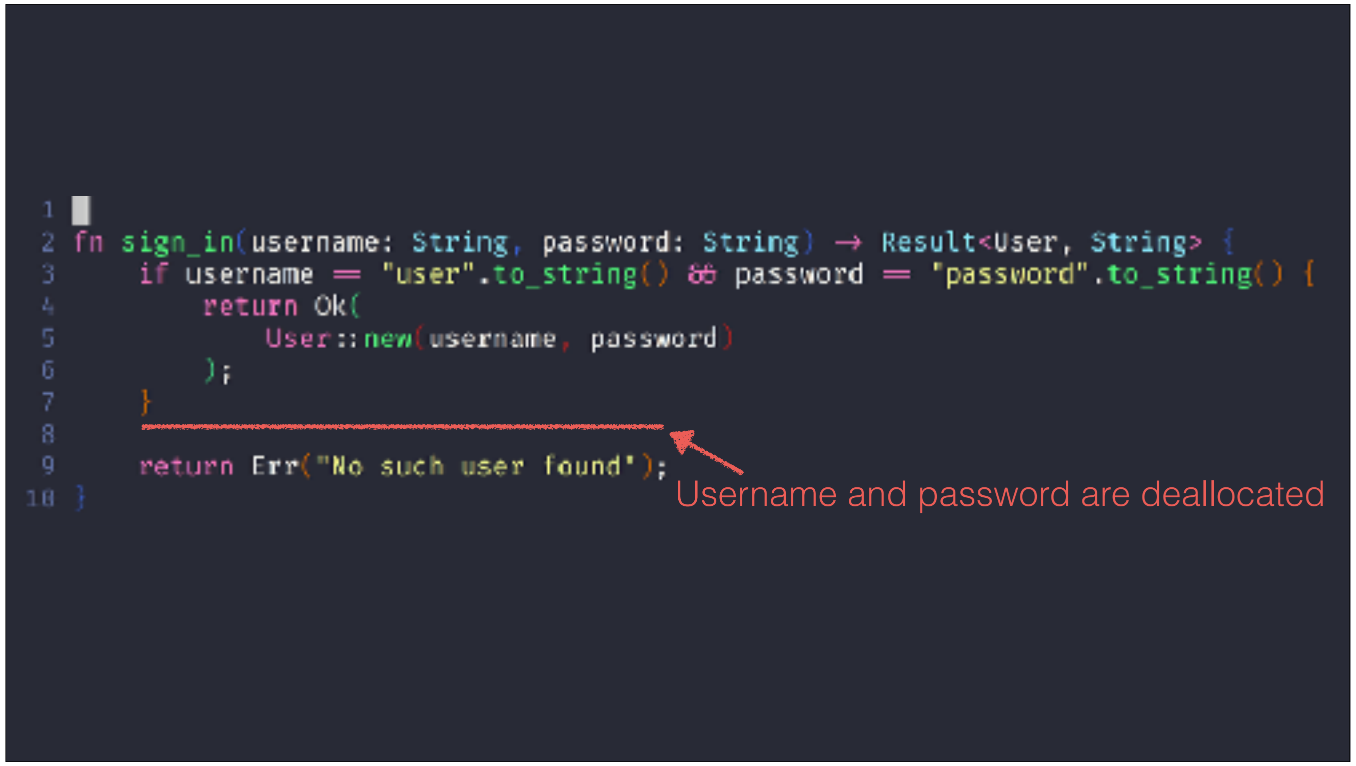
```

fn sign_in(username: String, password: String) → Result<User, String> {
    if username == "user".to_string() && password == "password".to_string() {
        return Ok(
            User::new(username, password)
        );
    }

    return Err("No such user found");
}
```

And it returns a result that will yield a user struct if successful, or it will yield a String if unsuccessful



```rust

fn sign_in(username: String, password: String) -> Result<User, String> {
    if username == "user".to_string() && password == "password".to_string() {
        return Ok(
            User::new(username, password)
        );
    }

    return Err("No such user found");
}
```

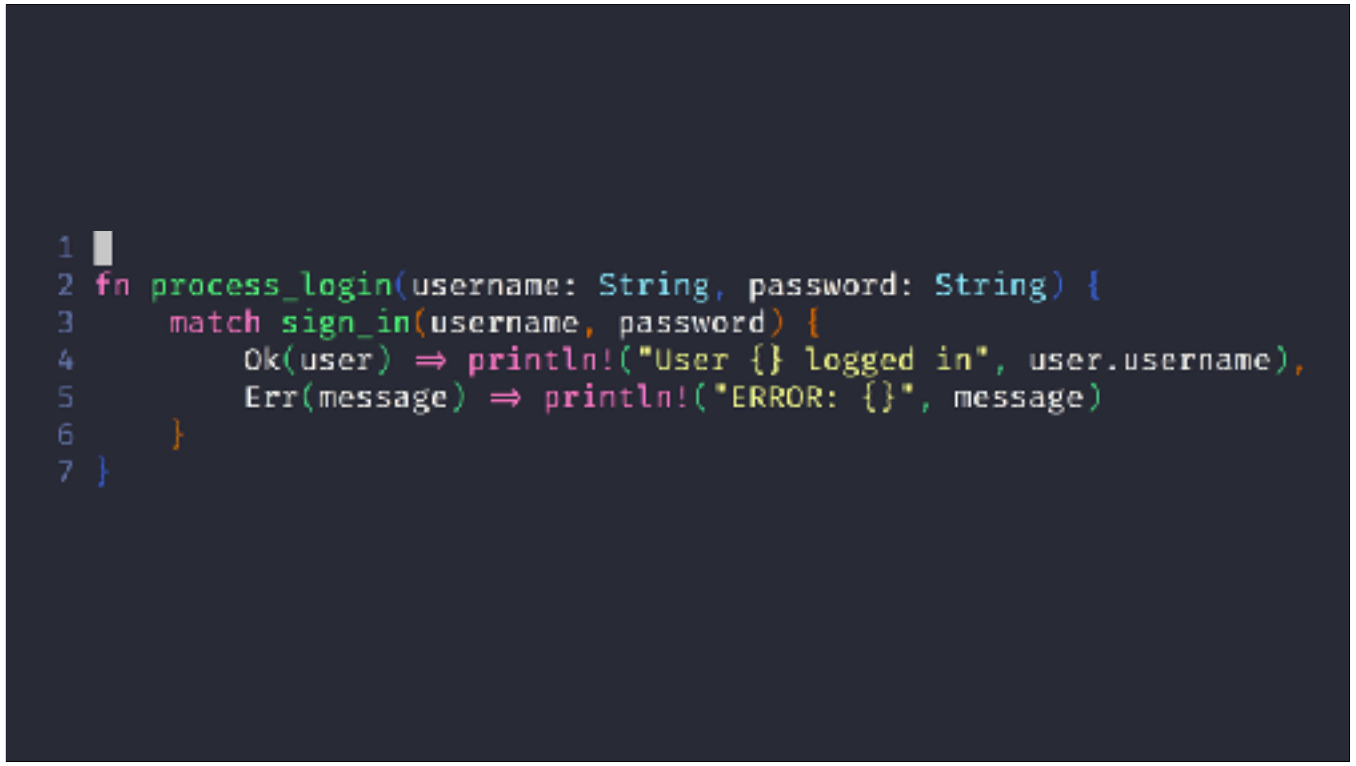
And to briefly explain lifetimes. Basically the compiler notices when a variable isn’t used anymore, and implicitly deallocates it after the line it’s last used



```rust

fn sign_in(username: String, password: String) -> Result<User, String> {
    if username == "user".to_string() && password == "password".to_string() {
        return Ok(
            User::new(username, password)
        );
    }

    return Err("No such user found");
}
```

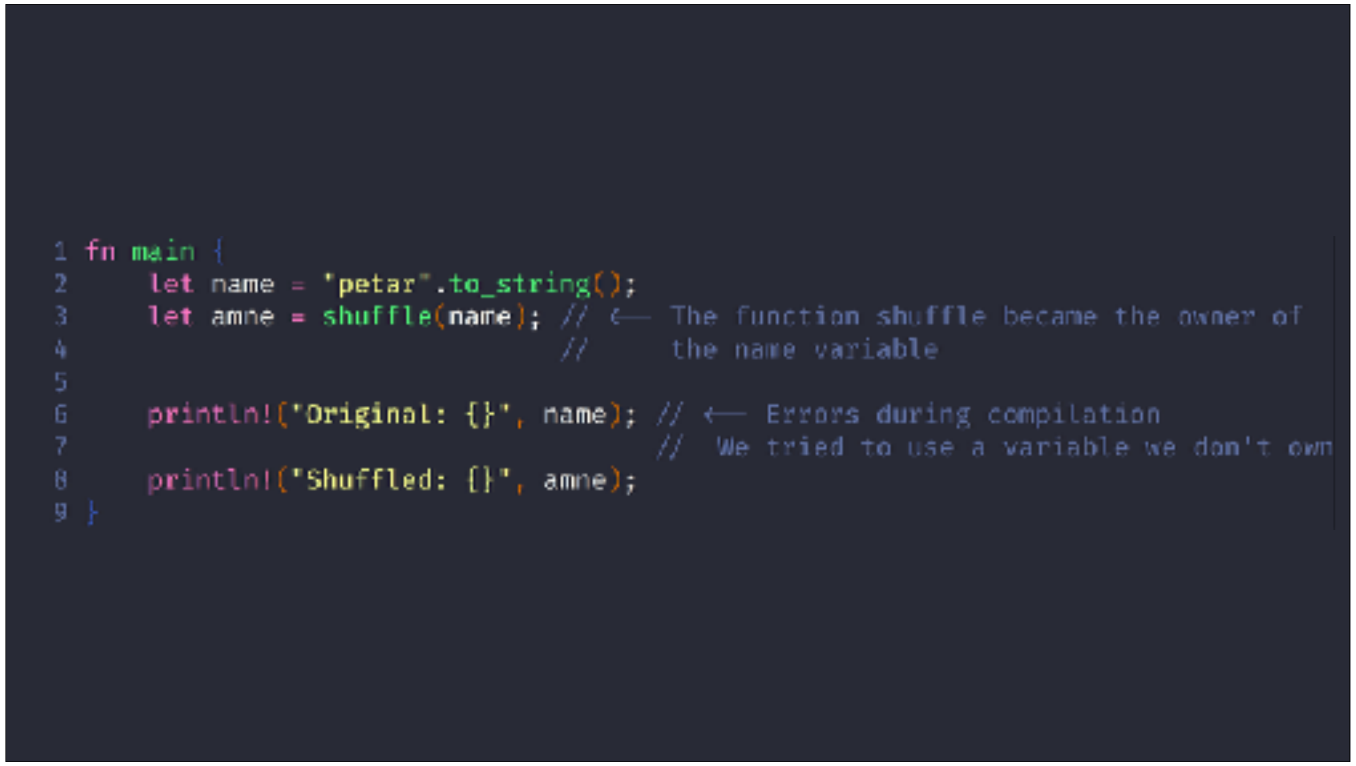
Username and password are deallocated

And to briefly explain lifetimes. Basically the compiler notices when a variable isn’t used anymore, and implicitly deallocates it after the line it’s last used



```rust

fn process_login(username: String, password: String) {
    match sign_in(username, password) {
        Ok(user) => println!("User {} logged in", user.username),
        Err(message) => println!("ERROR: {}", message)
    }
}
```

Then the function calling can just check if the result was an Ok or an Err value. This completely eliminates the need for NULL values, and throwing or raising errors. Thus making the programs flow more predictable. Not only that, but the compile will complain if you don’t process a returned result value. This pattern is used widely with primitives such as vectors, thus making it impossible to have any kind of non-memory safe operation.



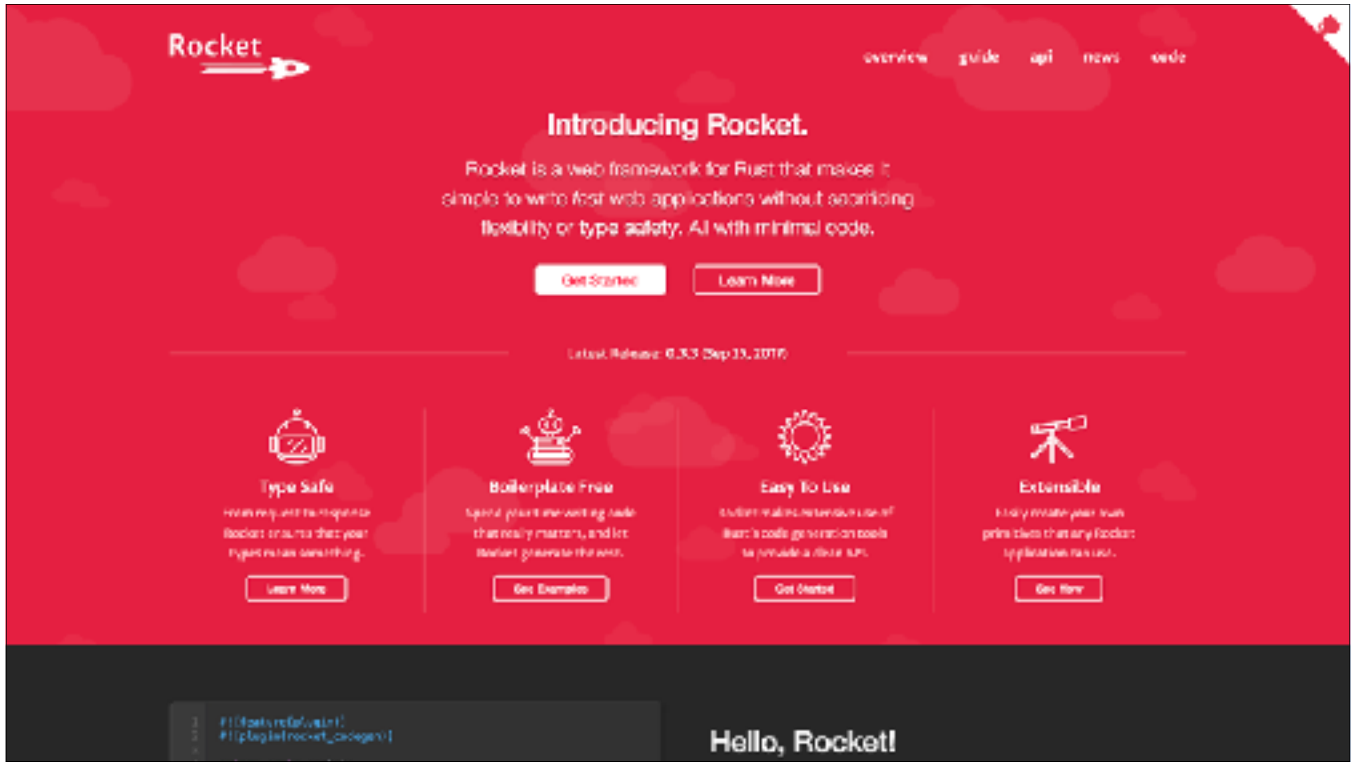
```
fn main {
    let name = "petar".to_string();
    let amne = shuffle(name); // ⟵ The function shuffle became the owner of
                              //     the name variable

    println!("Original: {}", name); // ⟵ Errors during compilation
                                    //  We tried to use a variable we don't own
    println!("Shuffled: {}", amne);
}
```

Ownership and borrowing aren't that east to fully explain, but here it is in a nutshell. You can think of it as if you are borrowing a book. Once you give a book to a friend you can't use it any more until they give it back to you. The same applies to this example. Ounce we gave the variable called "name" to the "shuffle" function we don't own it anymore, and therefore we can't use it. The compiler won't let us. This resolves all data race conditions. I won't go into lifetimes here, but believe me that they are pure magic that abolishes the need for a garbage collector.

03

WEB

Now that we know how Rust helps us avoid common bugs let’s check how it can help us with web development. So let’s start with the server!



Introducing Rocket.

Rocket is a web framework for Rust that makes it simple to write fast web applications without sacrificing flexibility or type safety. All with minimal code.

Latest Release: 0.3.3 (Sep 25, 2017)

Type Safe

From request to response Rocket ensures that your types mean something.

Boilerplate Free

Spend your time writing code that really matters, and let Rocket generate the rest.

Easy To Use

Rocket makes extensive use of Rust's code generation tools to provide a clean API.

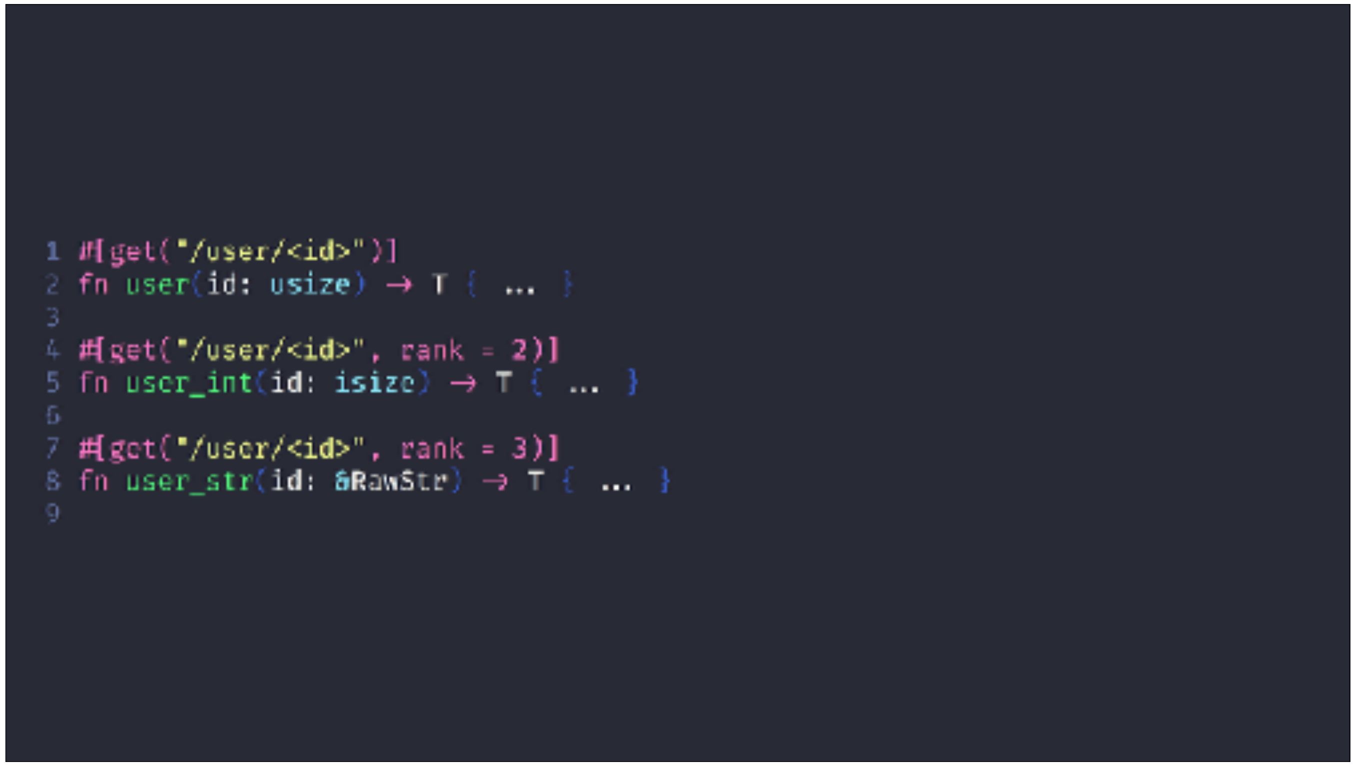
Extensible

Easily create your own primitives that any Rocket application can use.

```
#![feature(plugin)]
#![plugin(rocket_codegen)]
```

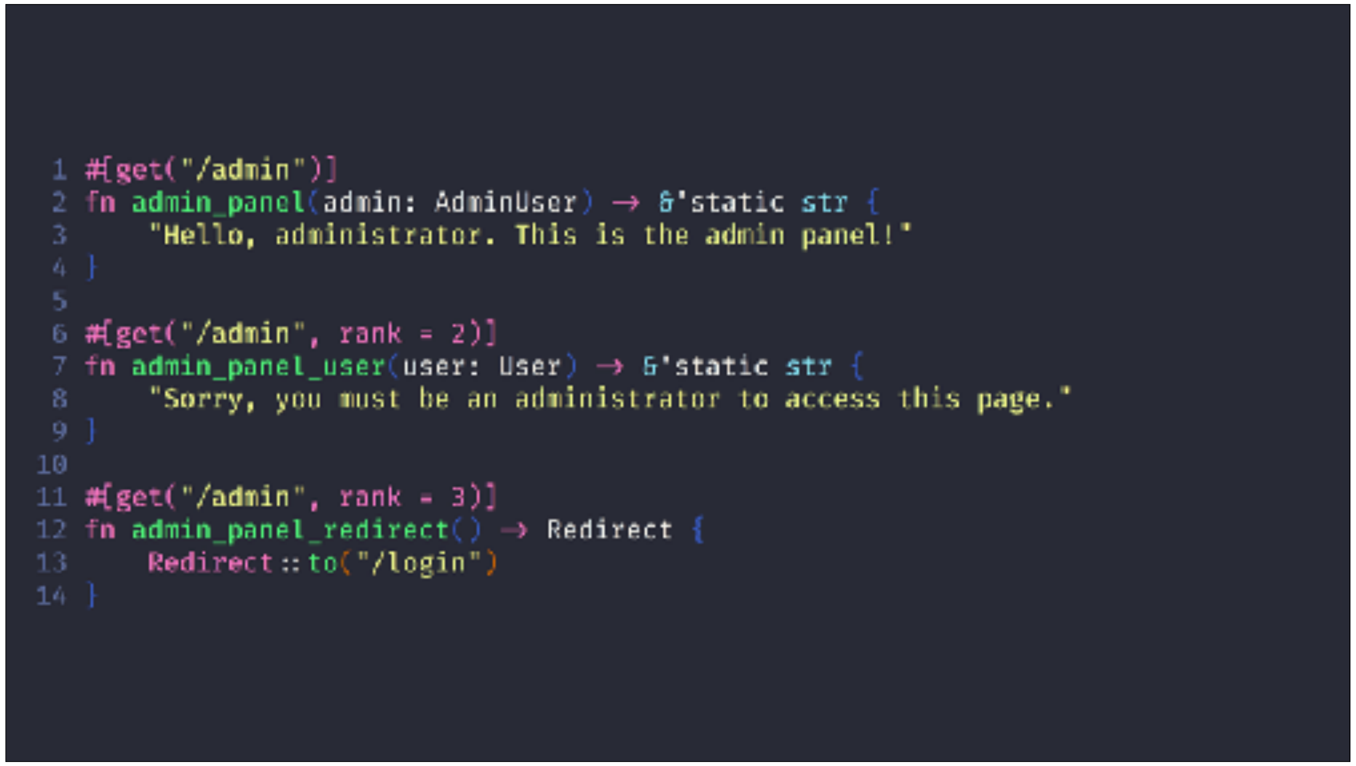
Hello, Rocket!

There are two major web servers for Rust. One is Iron, which offers a classic middleware-type server and a MVC architecture, and the other is Rocket. The tagline says it all. Rocket makes it easier to write safe web apps. It implements a kind-of-pattern-matching for requests which ensures that a function can respond to a request only if it is really able to process it.



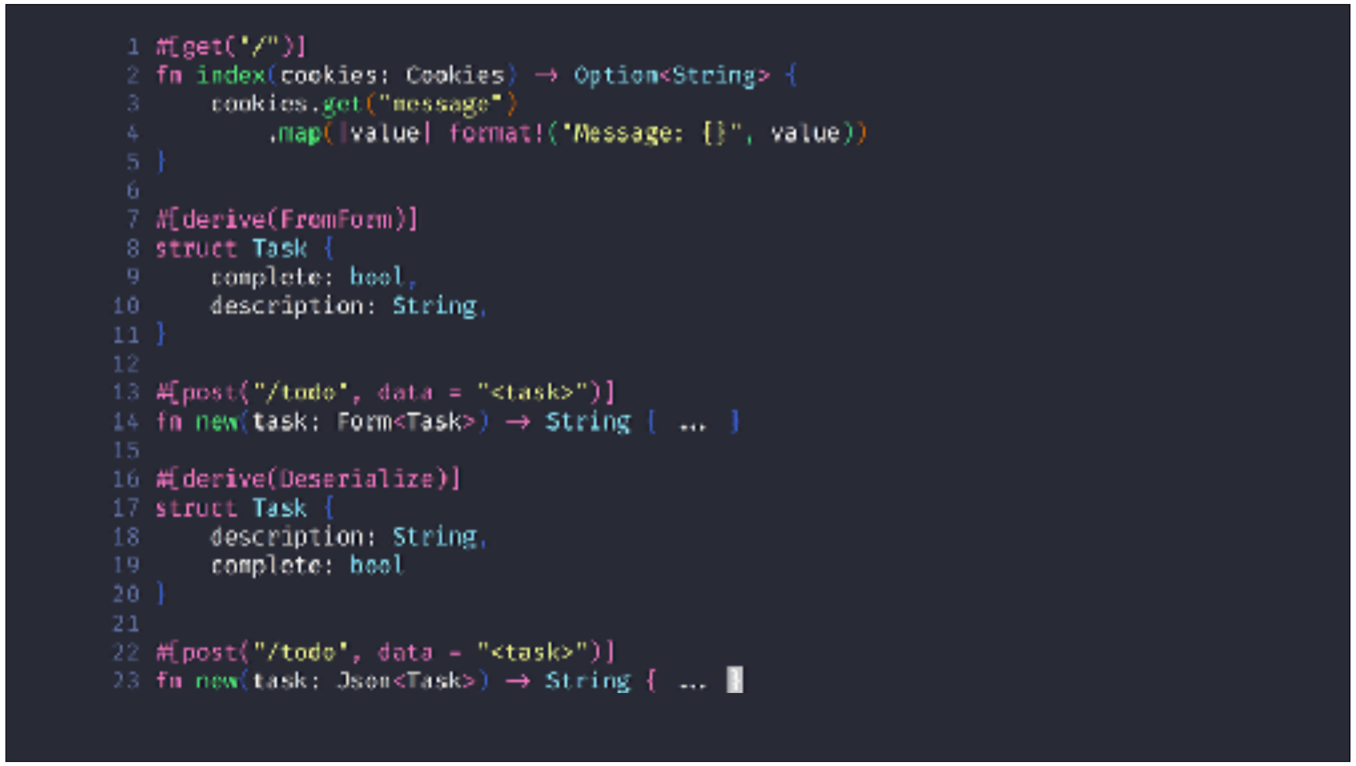
```rust
#[get("/user/<id>")]
fn user(id: usize) -> T { ... }

#[get("/user/<id>", rank = 2)]
fn user_int(id: isize) -> T { ... }

#[get("/user/<id>", rank = 3)]
fn user_str(id: &RawStr) -> T { ... }
```

This is the simplest example of this patter. As you can see all functions respond to the exact same endpoint, but expect different input types, and depending on what was passed as the ID the appropriate function will get called.



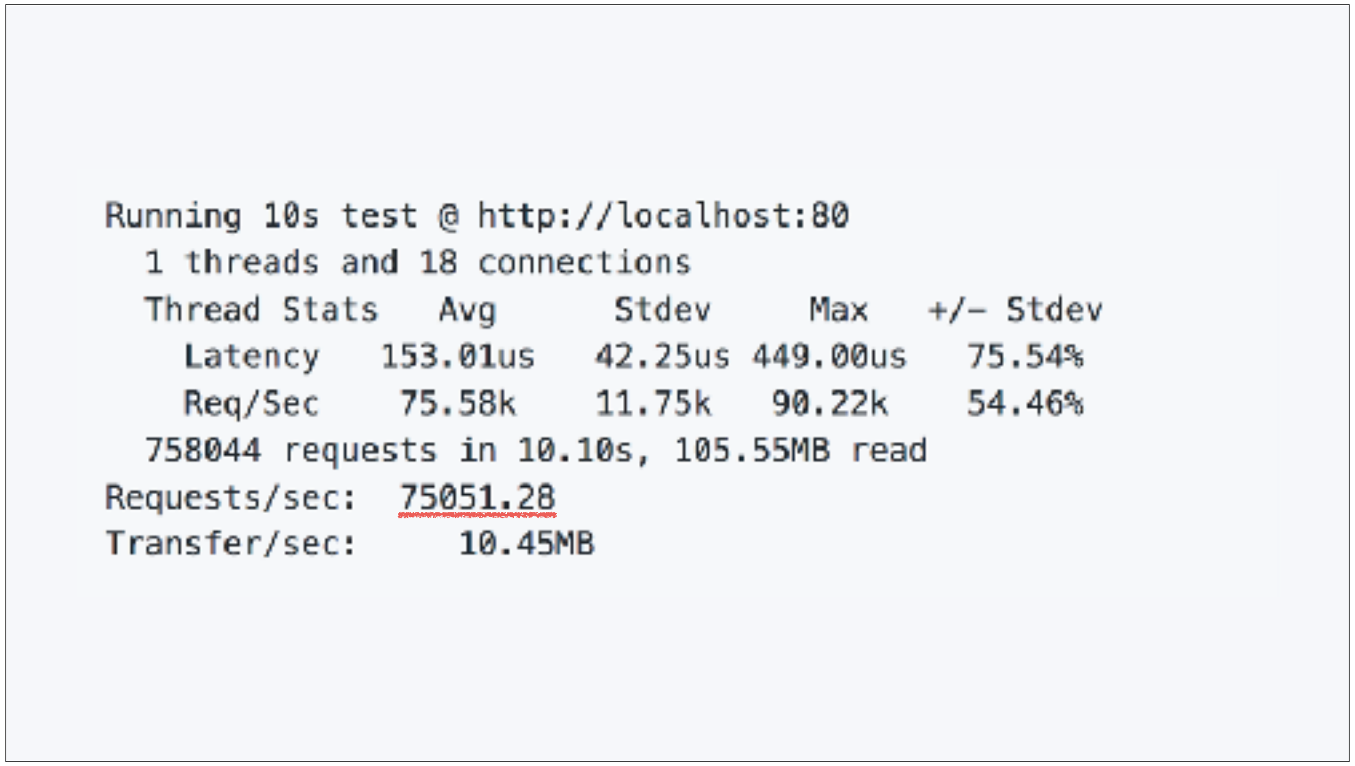
```rust
#[get("/admin")]
fn admin_panel(admin: AdminUser) -> &'static str {
    "Hello, administrator. This is the admin panel!"
}

#[get("/admin", rank = 2)]
fn admin_panel_user(user: User) -> &'static str {
    "Sorry, you must be an administrator to access this page."
}

#[get("/admin", rank = 3)]
fn admin_panel_redirect() -> Redirect {
    Redirect::to("/login")
}
```

Though the true power of this pattern lies in the ability to define custom types. This example shows how you can check the user's privileges and direct him to the appropriate function to process the request.



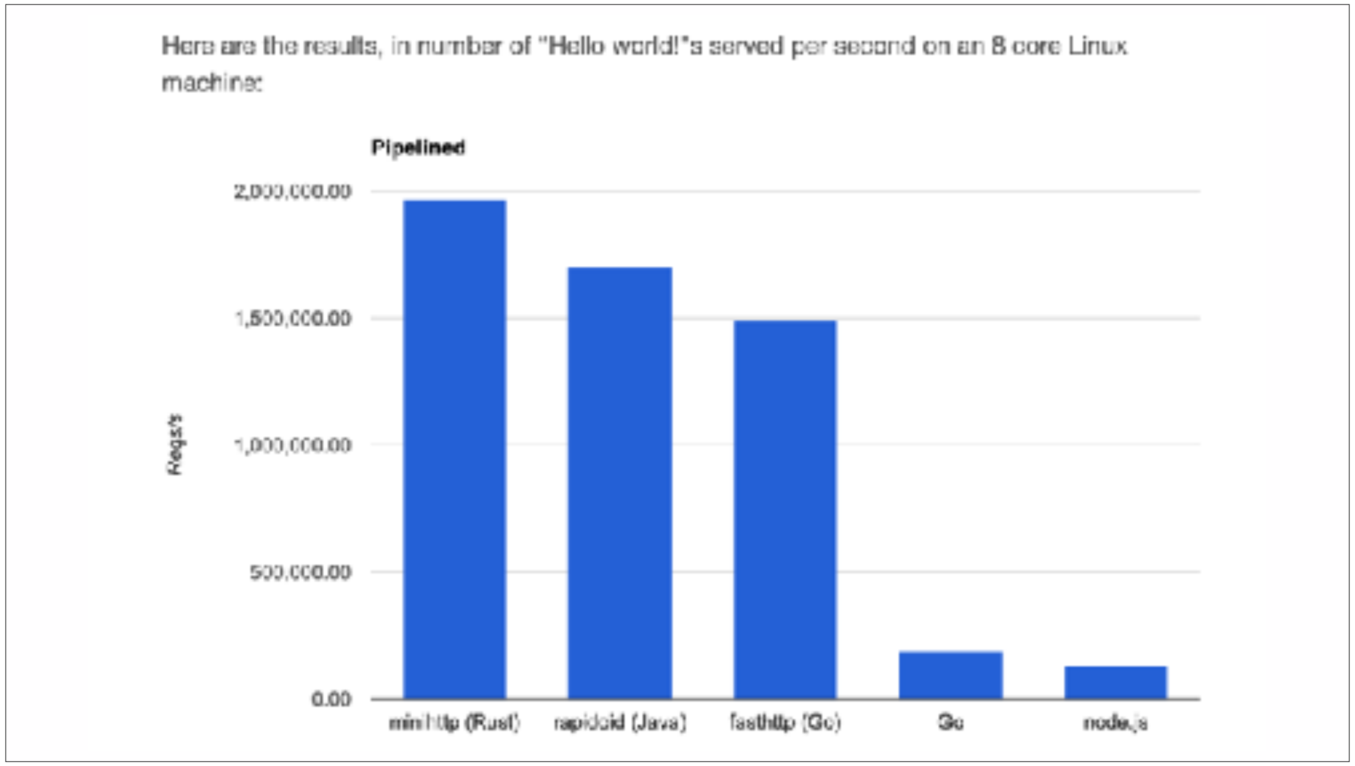
```
#[get("/")]
fn index(cookies: Cookies) → Option<String> {
    cookies.get("message")
        .map(|value| format!("Message: {}", value))
}

#[derive(FromForm)]
struct Task {
    complete: bool,
    description: String,
}

#[post("/todo", data = "<task>")]
fn new(task: Form<Task>) → String { ... }

#[derive(Deserialize)]
struct Task {
    description: String,
    complete: bool
}

#[post("/todo", data = "<task>")]
fn new(task: Json<Task>) → String { ... }
```

Rocket comes with a lot of pre-made data types for processing JSON data, Form data, Cookies, and other… This pattern has proven to be very effective at preventing various unwanted and unexpected behaviours.



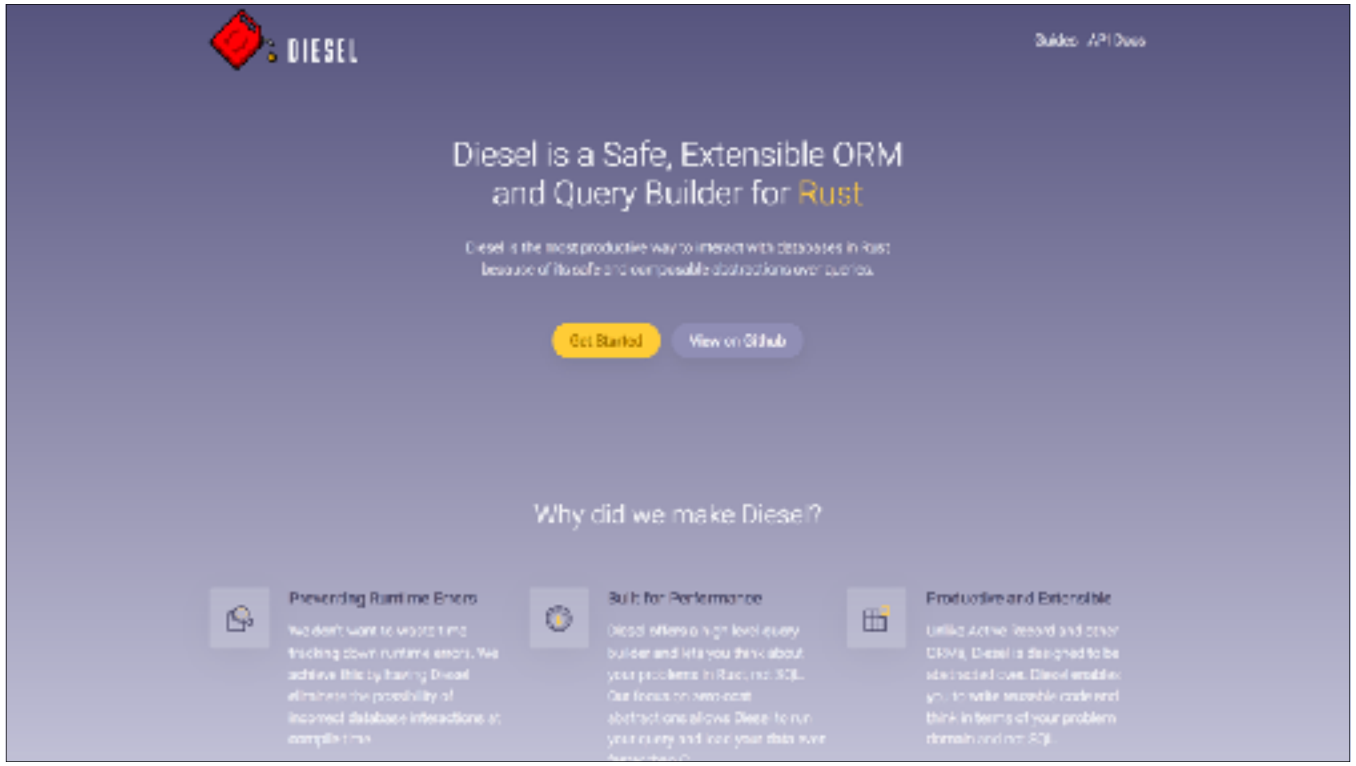
```
Running 10s test @ http://localhost:80
  1 threads and 18 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   153.01us   42.25us 449.00us   75.54%
    Req/Sec    75.58k    11.75k   90.22k    54.46%
  758044 requests in 10.10s, 105.55MB read
Requests/sec:  75051.28
Transfer/sec:     10.45MB
```

And, only because people ask for them so often, here is a benchmark. Rocket will give you around 70k requests per second on a single instance. Of course, always remember that all benchmarks are wrong and you should always do your own in your environment. Because something that’s fast for me doesn’t have to be fast for you. You may have different bottlenecks than I have, and so on…



Here are the results, in number of "Hello world!"s served per second on an 8 core Linux machine:

But be aware that if performance is what you are looking for then there is the Futures library which can serve up to two million requests per second.



Diesel is a Safe, Extensible ORM and Query Builder for Rust

Diesel is the most productive way to interact with databases in Rust because of its safe and composable abstractions over queries.

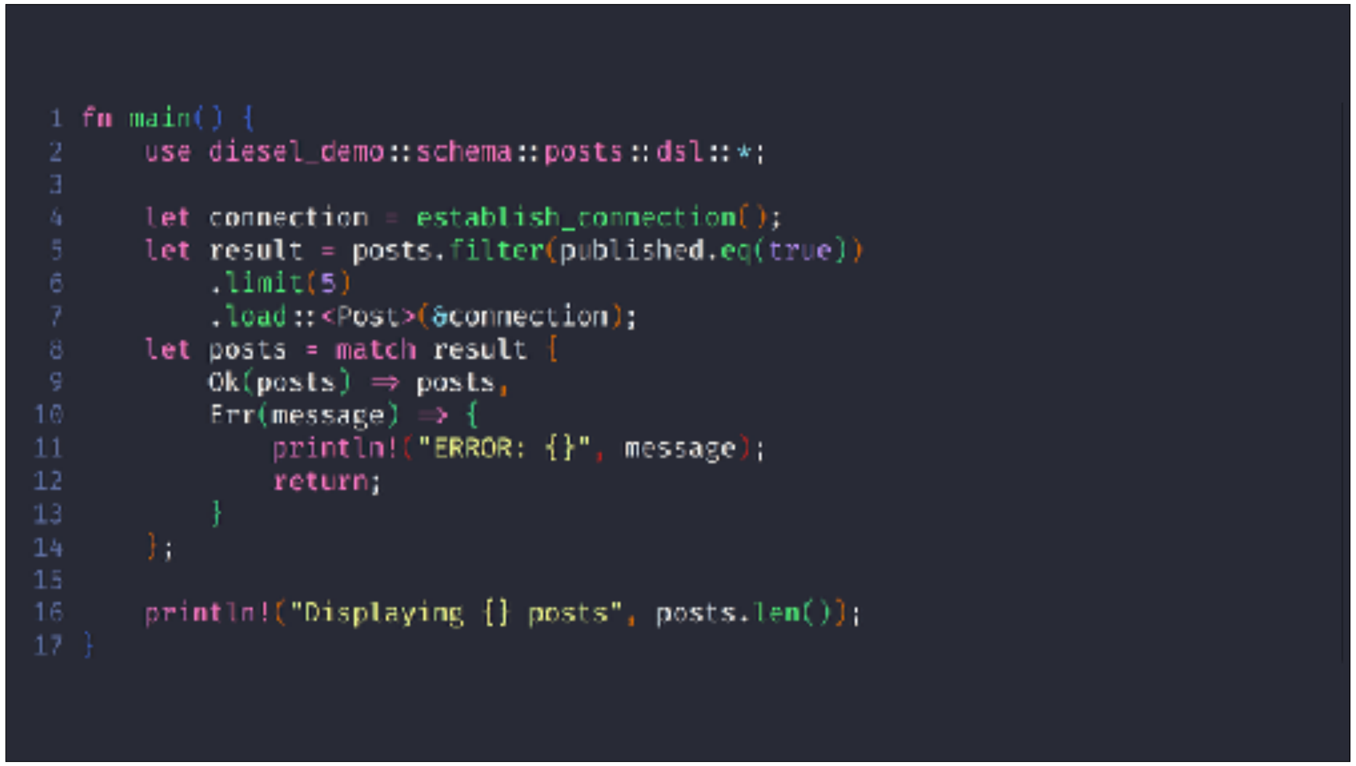
Why did we make Diesel?

Preventing Runtime Errors
We don't want to waste time tracking down runtime errors. We achieve this by having Diesel eliminate the possibility of incorrect database interactions at compile time.

Built for Performance
Diesel offers a high level query builder and lets you think about your problems in Rust, not SQL. Our focus on zero-cost abstractions allows Diesel to run your query and load your data even faster than C.

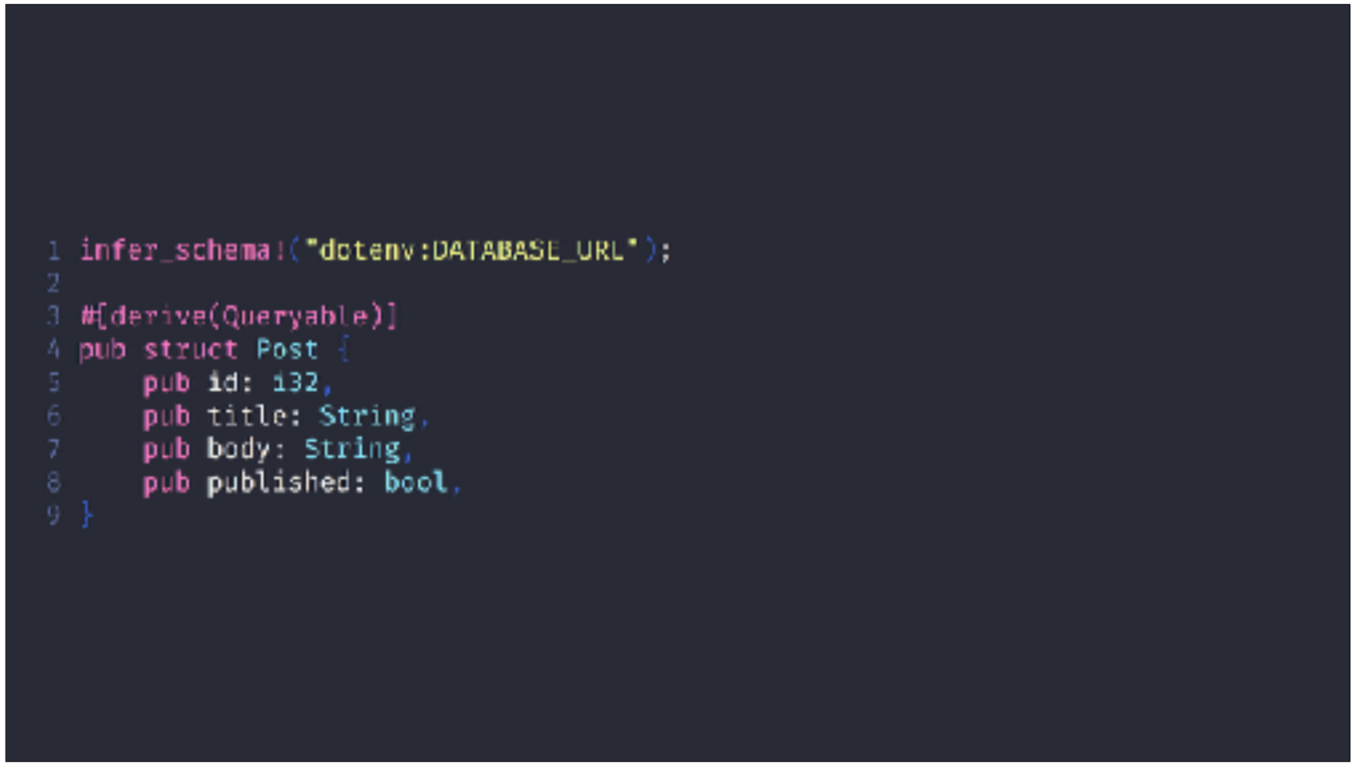
Productive and Extensible
Unlike Active Record and other ORMs, Diesel is designed to be abstracted over. Diesel enables you to write reusable code and think in terms of your problem domain and not SQL.

Now that we can get data in and out of our app we need a way to store it. Or in other words, we need an ORM. The most popular one, by far, is Diesel.



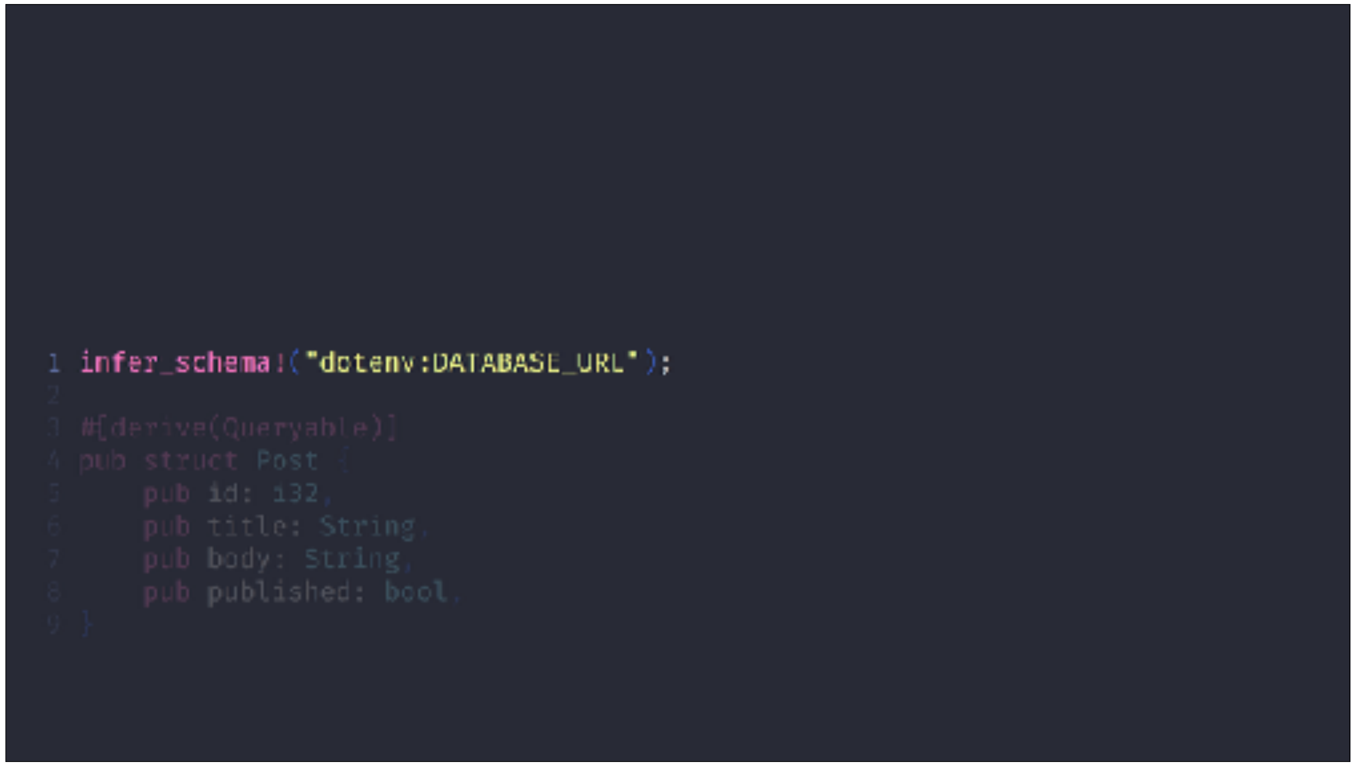
```rust
fn main() {
    use diesel_demo::schema::posts::dsl::*;

    let connection = establish_connection();
    let result = posts.filter(published.eq(true))
        .limit(5)
        .load::<Post>(&connection);
    let posts = match result {
        Ok(posts) => posts,
        Err(message) => {
            println!("ERROR: {}", message);
            return;
        }
    };

    println!("Displaying {} posts", posts.len());
}
```

Diesel provides you with the bog standard feature set that you would expect from an ORM. You have a way to retrieve, insert and update data in the database. It can be used in conjunction with most SQL databases.



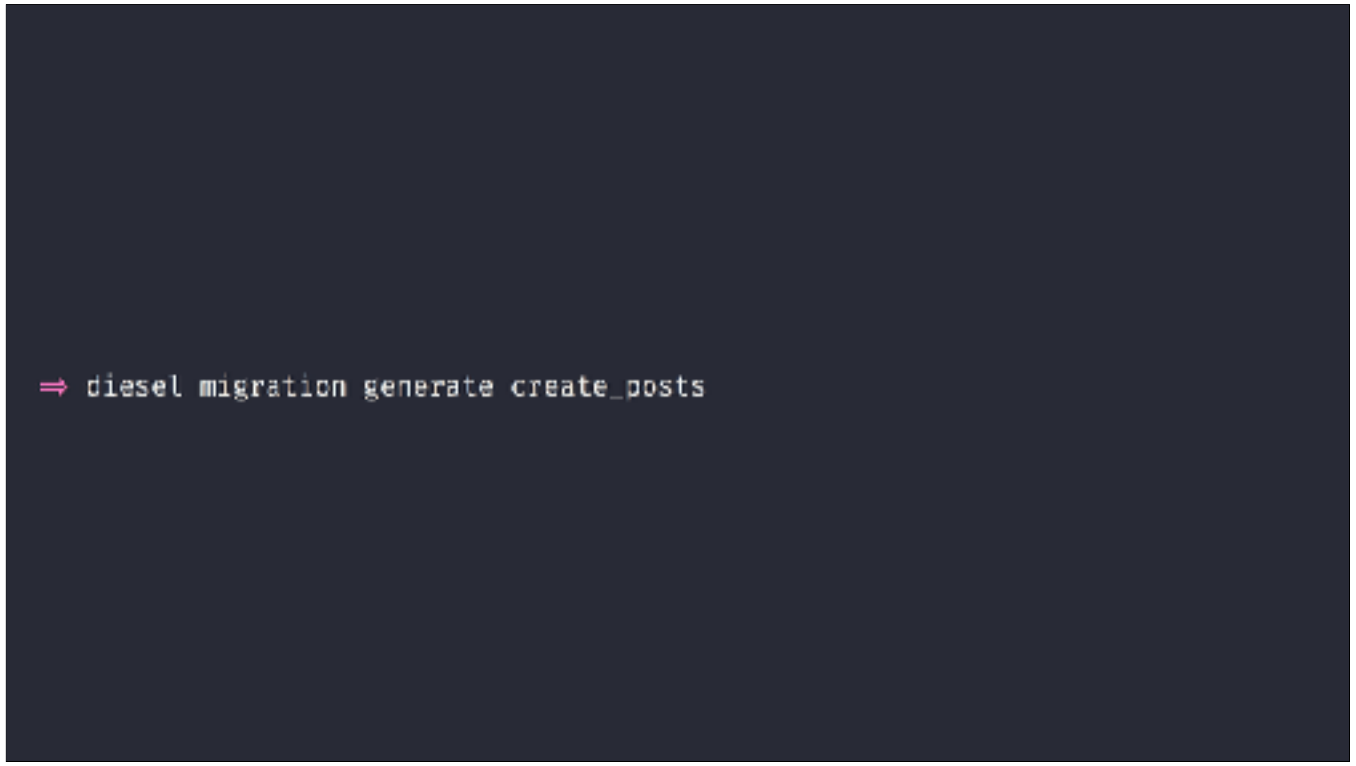
```rust
infer_schema!("dotenv:DATABASE_URL");

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

The cool part air all the other features, like schema inference.



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

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

If you type this macro, during compilation, Diesel will connect to the database and infer a schema from it, then it will enforce that schema on the structs that represent your models in the database. Basically it will complain if you don’t have a field that’s present in the database, or if you represent a field with the wrong type.



```
⇒ diesel migration generate create_posts
```

And it comes with a nice CLI tool to make your life a bit easier

Iron

Tokio

Mio

Rayon

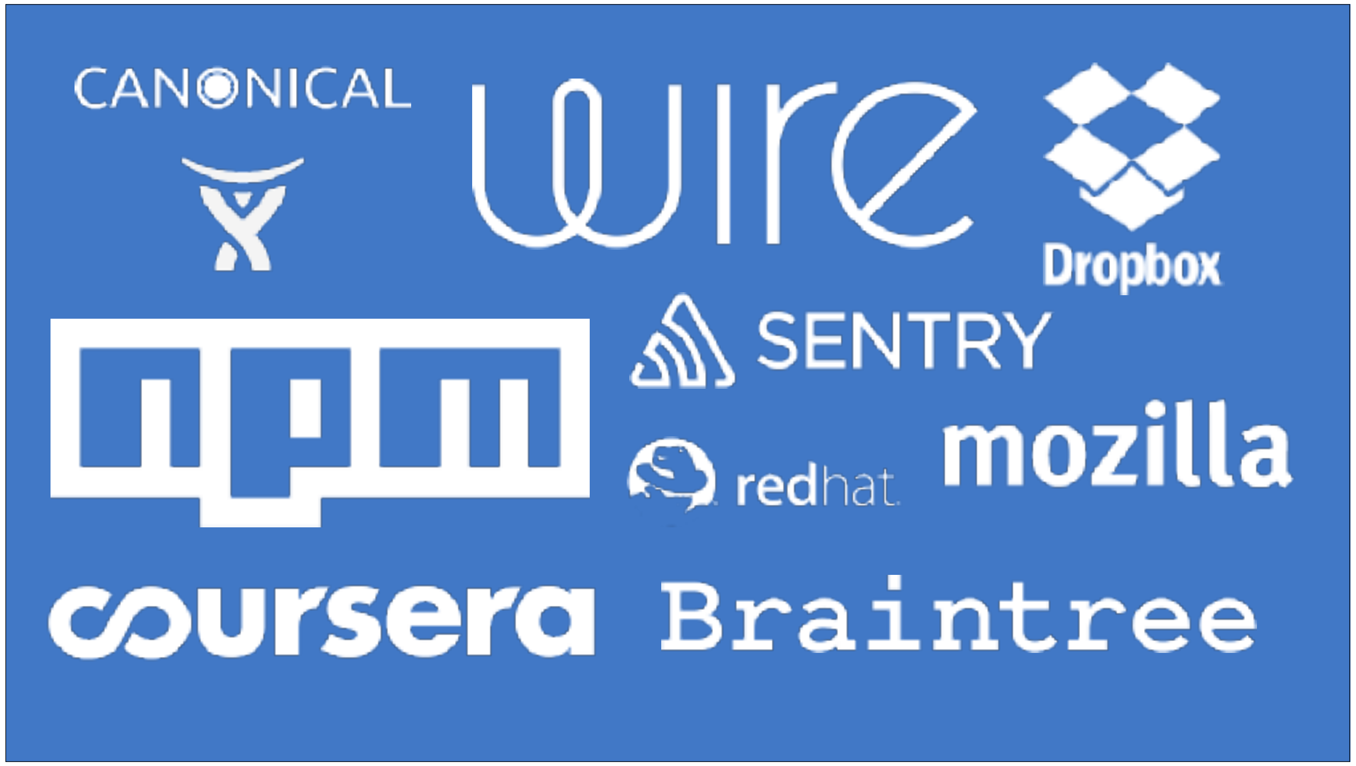
Futures

Grpc-rust

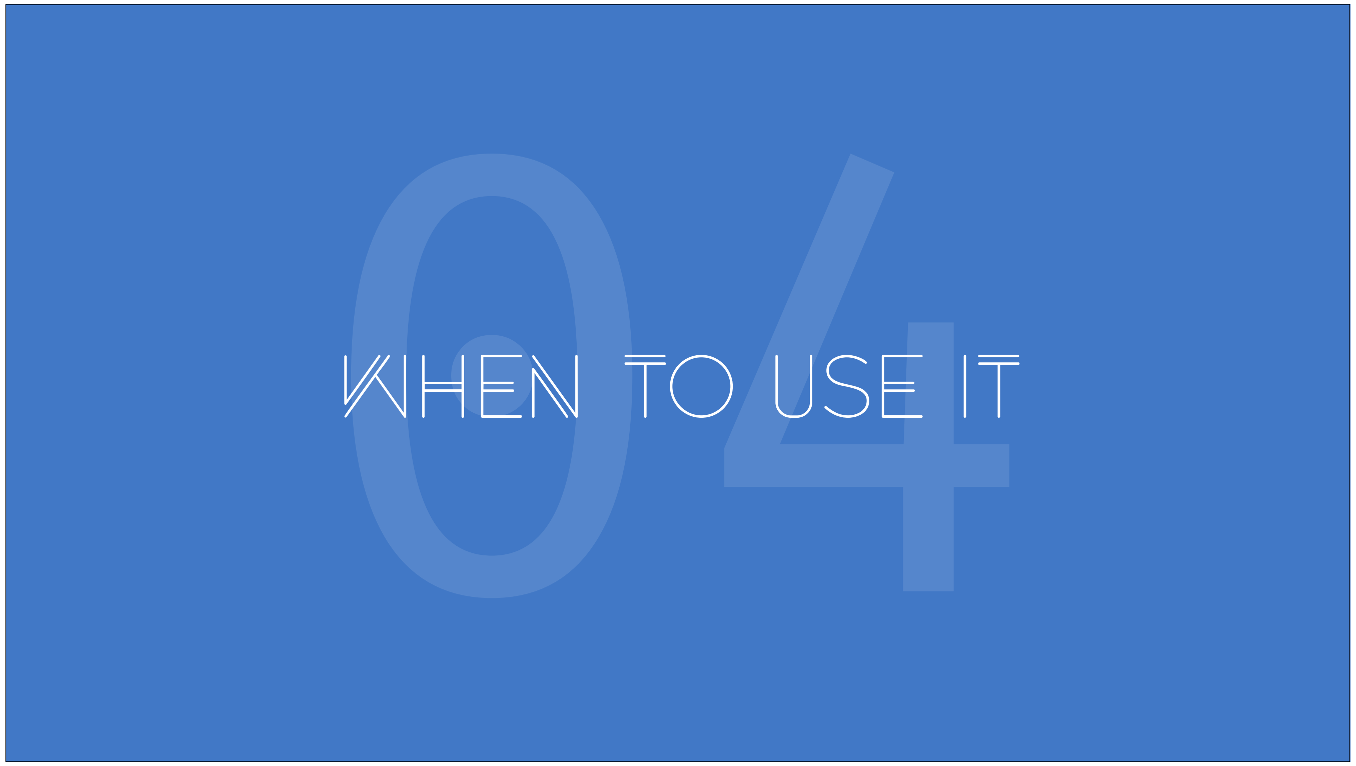
Juniper

Rust-ampq

But there are many more awesome libraries. Iron, a MVC web framework. Tokio, a fast networking library. Mio, a super fast, and safe IO library. Rayon, a library that builds parallel computation into standard primitives. Futures, which I talked about briefly. Juniper, a GraphQL implementation. GRPC and AMPQ are supported. The ecosystem has grown and is mostly mature by now.



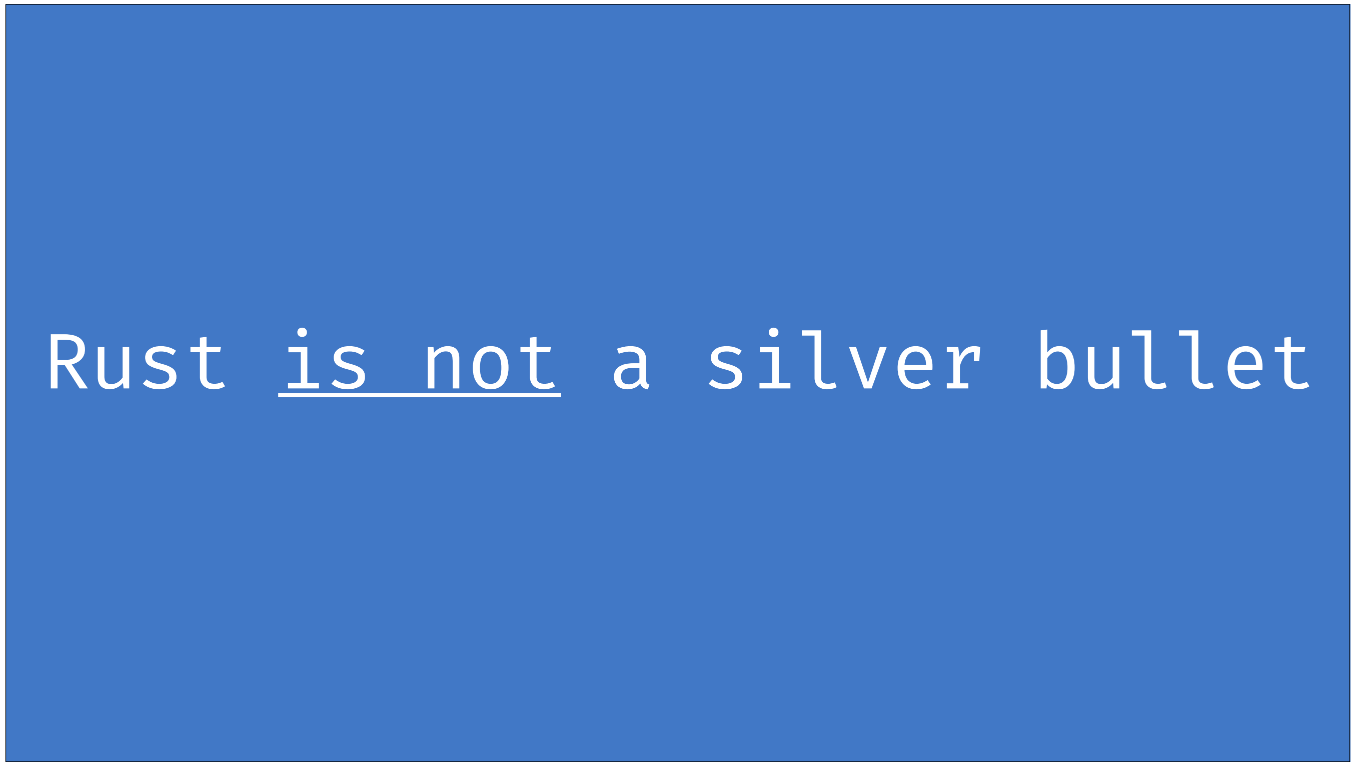
And here are some companies that use Rust in production. Just to show that it’s a mature technology.



04

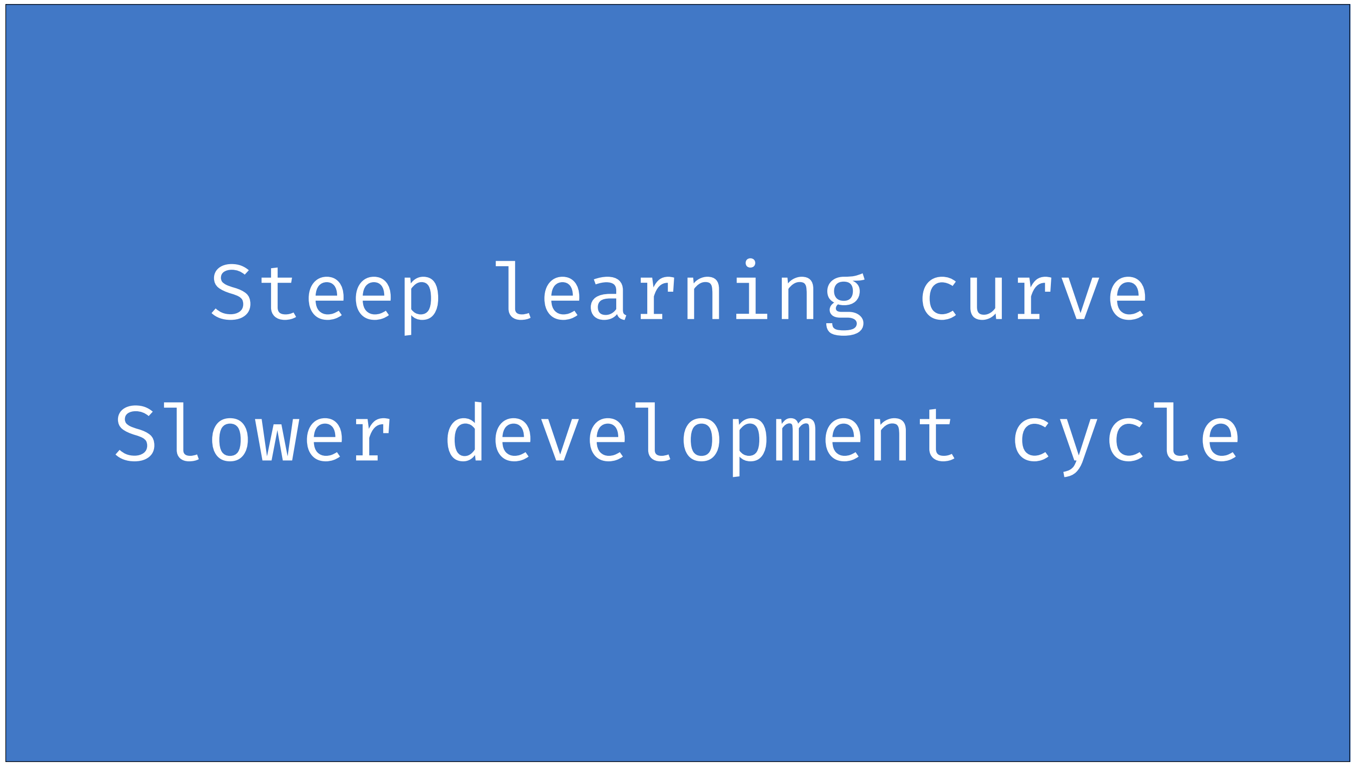
WHEN TO USE IT

But when do we use Rust.



Rust is not a silver bullet

Rust isn’t a silver bullet. Though you can’t make stupid mistakes and bugs, you can still write algorithmical bugs. Nothing can save you from that, that’s why you have a job after all.



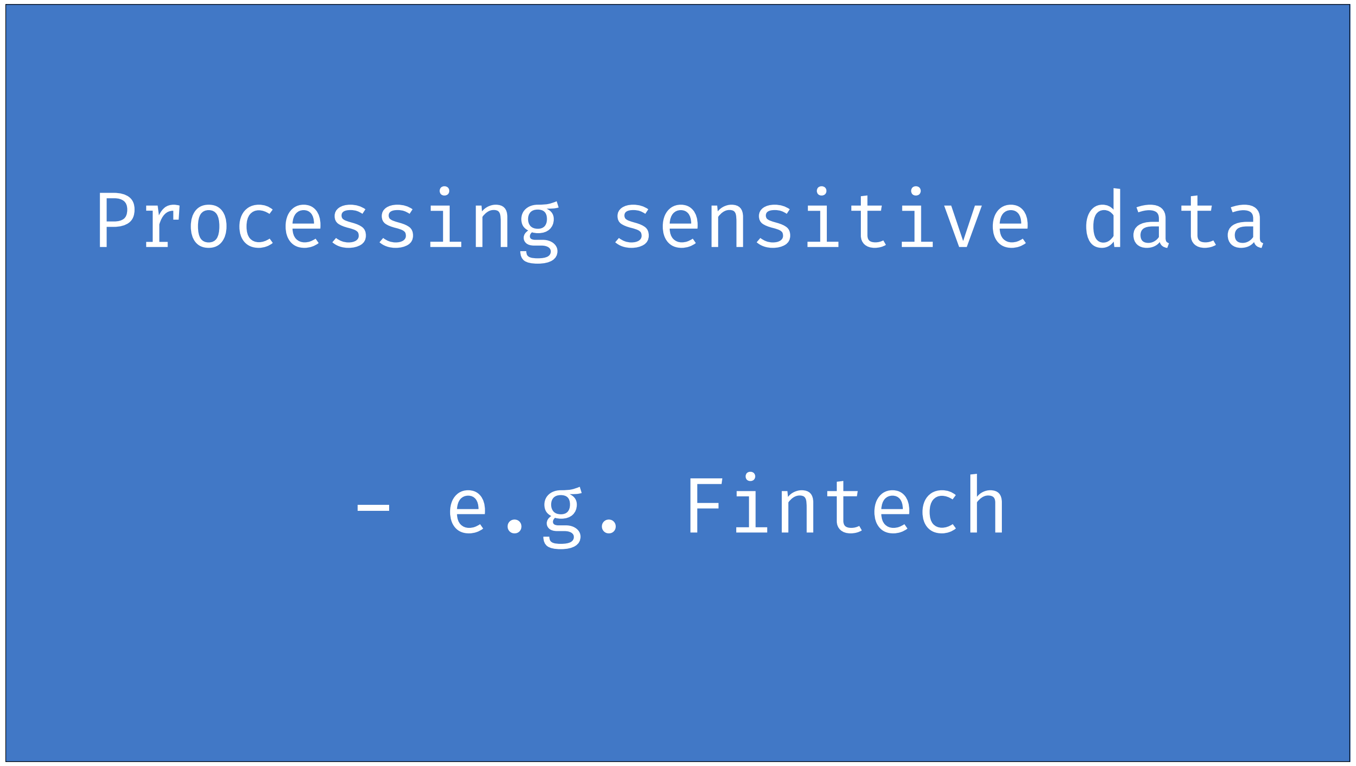
Steep learning curve

Slower development cycle

Rust has a steep learning curve, don’t expect to be proficient in Rust after you read the book. Ownership and borrowing are things that still bite me some times, even after two years working with the language. And your development cycle will get a bit slower. From my personal log I can tell you that any project I do in Rust is about 40% slower to develop than with Ruby. Though I also spend a lot less time fixing bugs.

But…

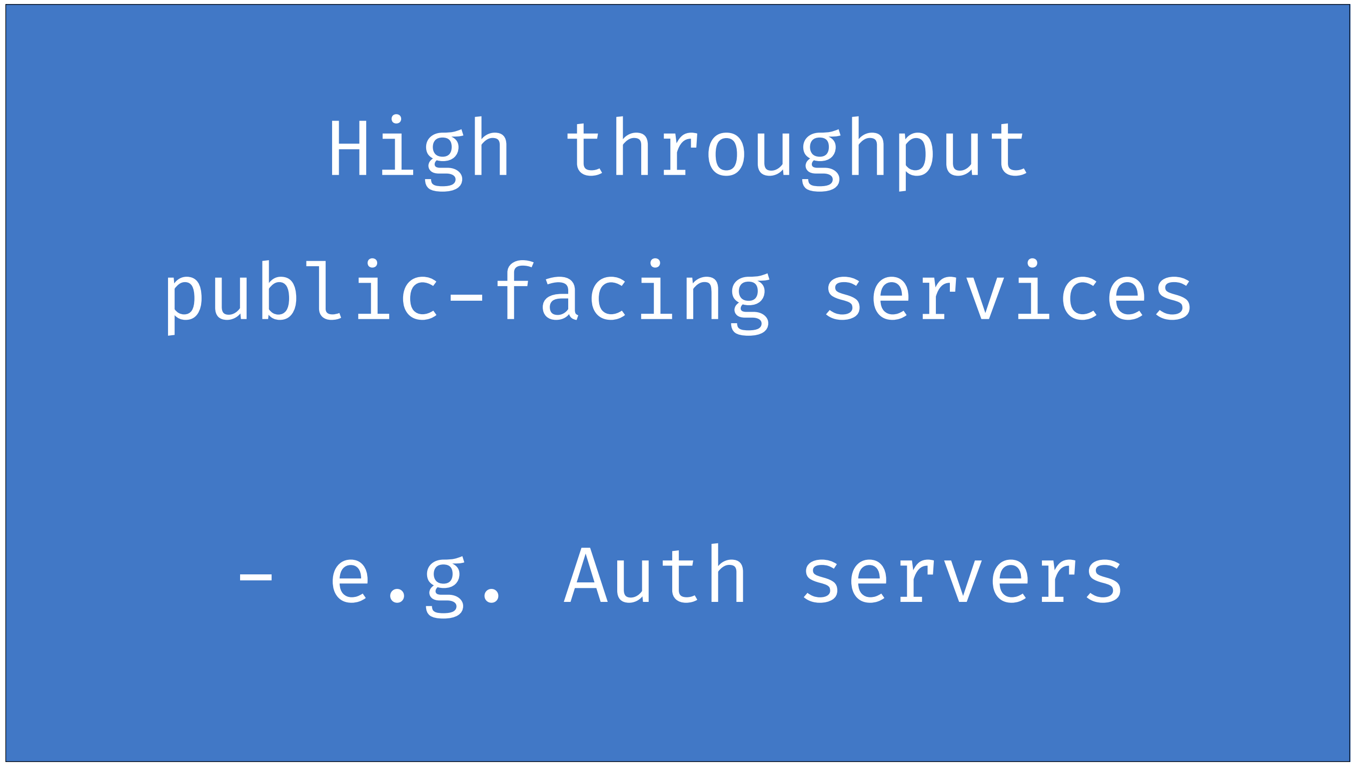
But…



Processing sensitive data

- e.g. Fintech

It excels in applications that have to process sensitive data such as Finch applications

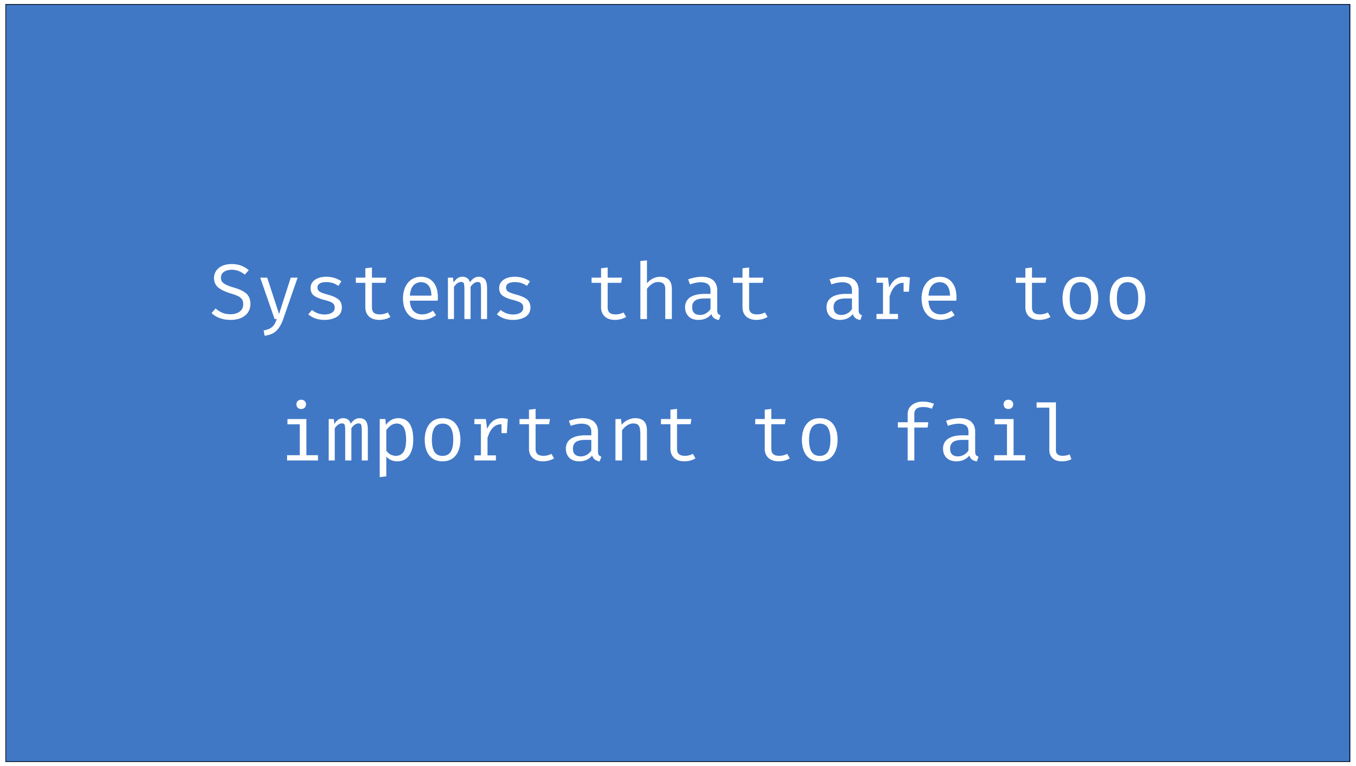


High throughput

public-facing services

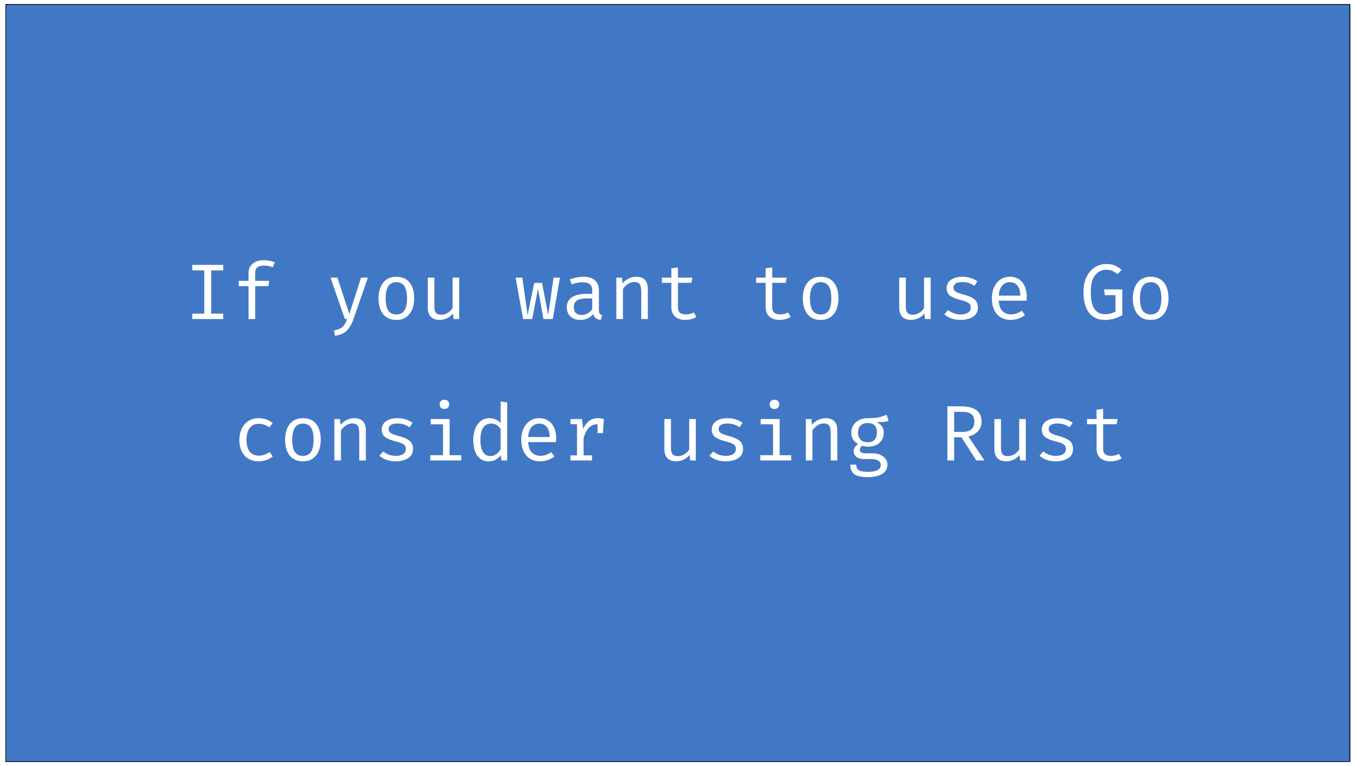
- e.g. Auth servers

High throughput scenarios, such as auth-servers, or database frontends.



Systems that are too important to fail

In general, it’s a no-brainer for systems that should never fail.



If you want to use Go
consider using Rust

Also, a good rule of thumb is that if you want to use Go for something, you should also think about Rust.

QUESTIONS

github.com/stankec
@monorkin
hi@stanko.io

Leave feedback on:
https://goo.gl/rwXEXd

Thank you for your patience. I know this was a lot in a short amount of time.
But if we have time I will accept questions now.
You can contact me via the methods printed on the screen.
And please take a second and rate me on joined.in, that’s the link at the bottom.