PAVEX

Re-imagining Rust backend development

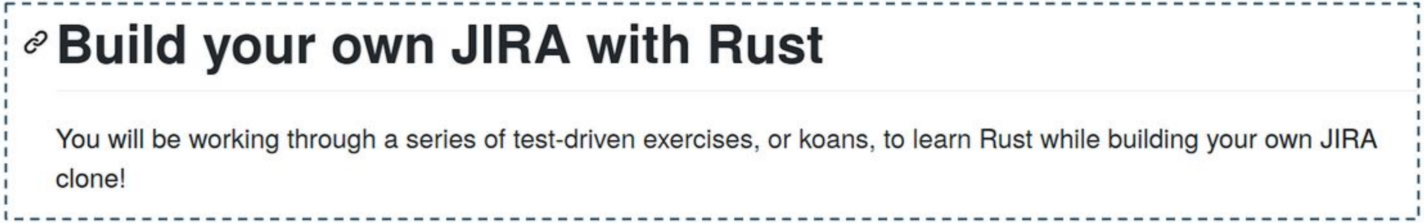
Luca Palmieri

@algo_luca

Luca Palmieri

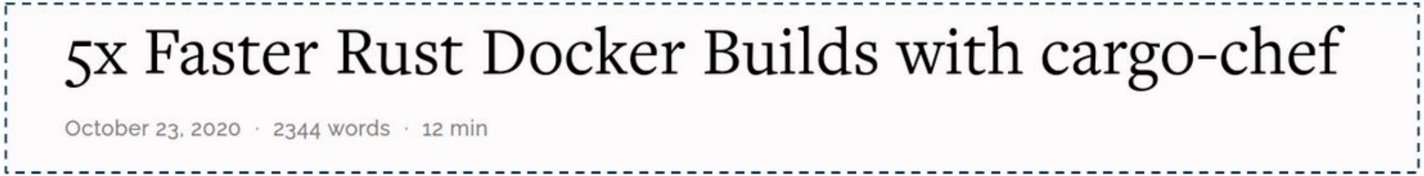
Principal Engineering Consultant
@ Mainmatter

@algo_luca
https://lpalmieri.com



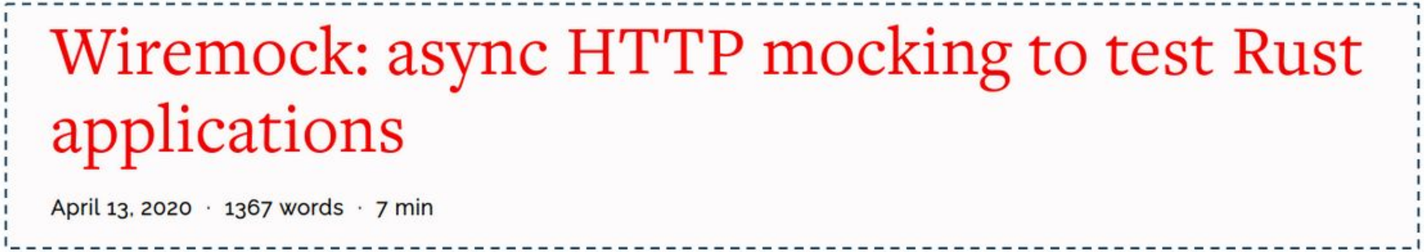
Build your own JIRA with Rust

You will be working through a series of test-driven exercises, or koans, to learn Rust while building your own JIRA clone!



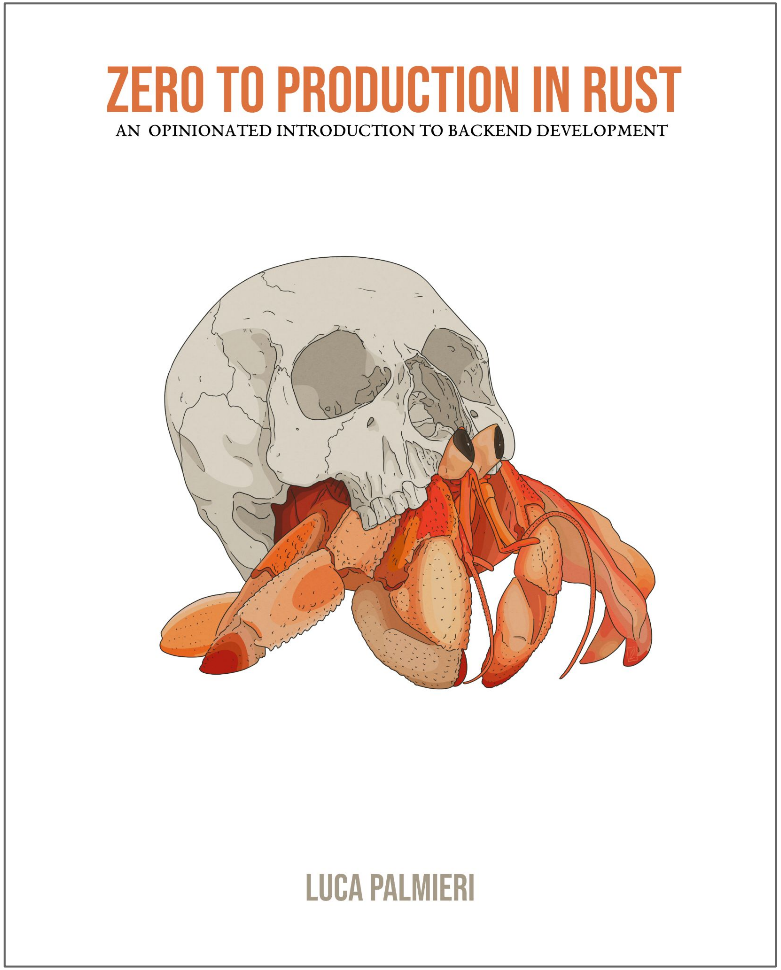
5x Faster Rust Docker Builds with cargo-chef

October 23, 2020 · 2344 words · 12 min



Wiremock: async HTTP mocking to test Rust applications

April 13, 2020 · 1367 words · 7 min



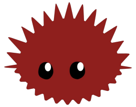
ZERO TO PRODUCTION IN RUST

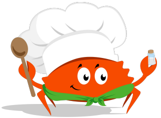
AN OPINIONATED INTRODUCTION TO BACKEND DEVELOPMENT

LUCA PALMIERI

https://zero2prod.com

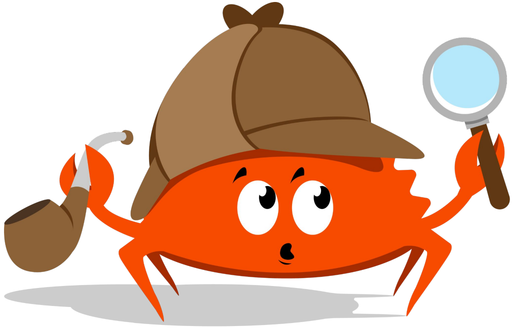
Agenda


Rust: are we backend yet?


Pavex

A look under the hood

Anatomy of a backend

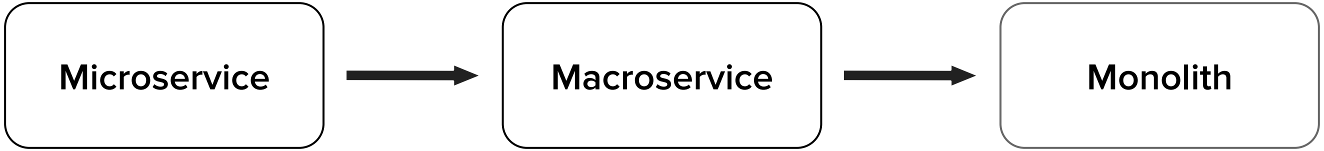


What does a modern backend look like?

It depends™

There is a varied zoology,
depending on the **dimensions** we are looking at

By size



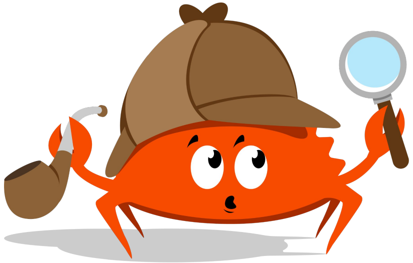
Microservice → Macroservice → Monolith

By interface

REST JSON

gRPC

GraphQL



By lifecycle

Long-lived
(“serverful”)

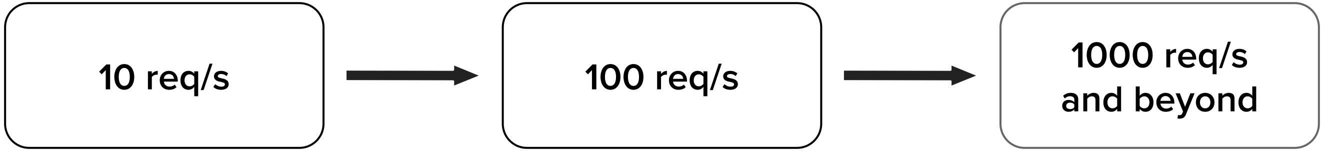
Short-lived
(serverless)

By client

Internal-facing

External-facing

By volume

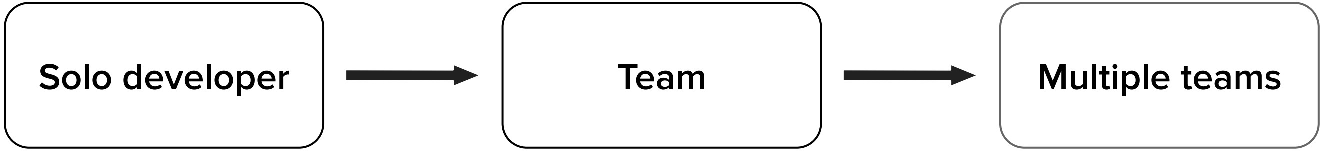


10 req/s

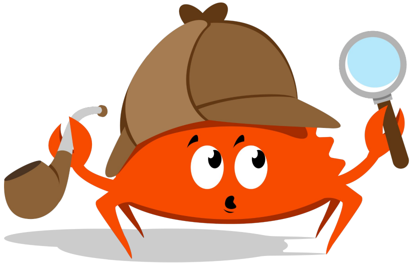
100 req/s

1000 req/s
and beyond

By team size



Solo developer → Team → Multiple teams



They all share **some** challenges,

but **each combination has its unique requirements**

To make things worse,

projects don’t stand still

A prototype becomes successful

You hire **a bigger team** to keep up with
an application that’s **growing in complexity**

Your company is acquired

All your services now need to **migrate to gRPC**, the technology your acquirer has standardized on

Your product is growing like crazy

You need to **migrate** your key workloads
away from serverless to keep costs under control

You must be careful when choosing
the **technology stack** you’ll be building on

Your foundation must be **specialised enough** to unlock productivity

But **flexible enough**

to evolve with your requirements

Rust: are we backend yet?

Is **Rust** a good choice

for building backend systems?

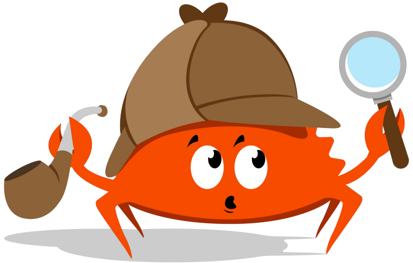
Yes*

Performance

✅

Team collaboration

✅



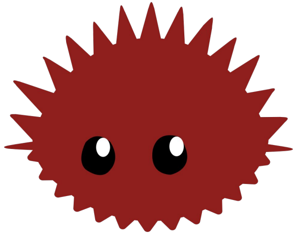
Supported platforms

✅

Nonetheless,

Rust is not a mainstream choice

for backend development



Rust has seen **limited success**

in **some** backend **niches**

Rust's backend niches:

→ **High performance requirements**

Rust’s backend niches:

→ High performance requirements

→ **High infrastructure footprint**

Rust’s backend niches:

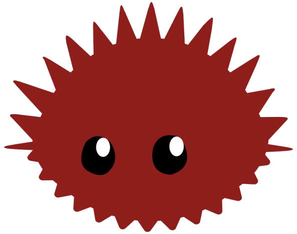
- → High performance requirements
- → High infrastructure footprint
- → **High reliability requirements**

“People come to Rust for its **performance**,
but they stick around for its **reliability**”

What’s **holding us back** from mainstream usage?

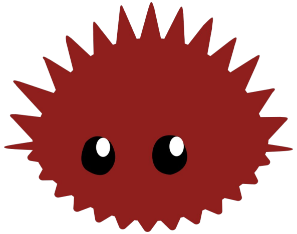
Rust’s weaknesses for backend:

➔ Limited talent pool with **professional** experience

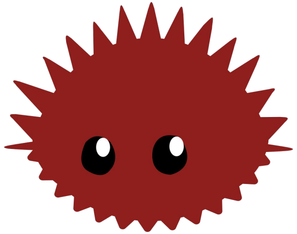


Rust’s weaknesses for backend:

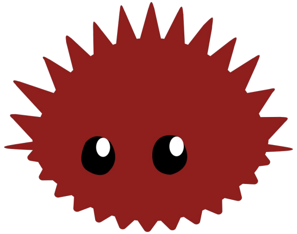
- → Limited talent pool with professional experience
- → **A Lego-like ecosystem**



async/await was stabilised 4 years ago,
at the end of 2019

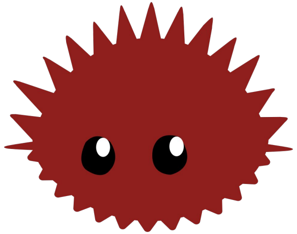


On those foundations, in 4 years,
the Rust community has built
a vast collection of high-quality libraries

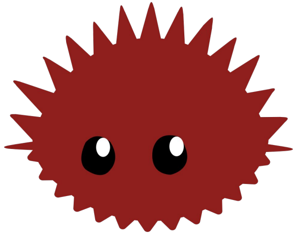


The collection is perhaps...

too vast

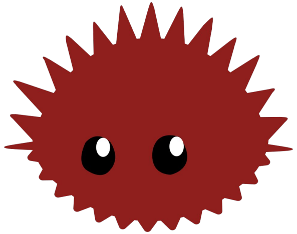


Beginners are **overwhelmed**



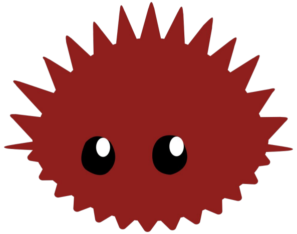
Too many choices to make,

too early in the journey

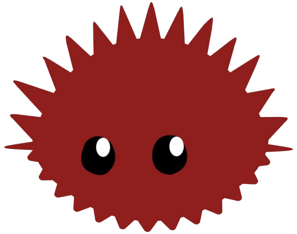


Complexity compounds:

each library needs to be good enough on its own
and interoperate with all the other ones you chose

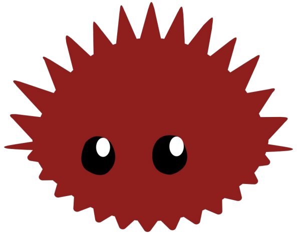


We need a **curated set of crates**,
with a **coordinated versioning policy**
and a **comprehensive feature set**

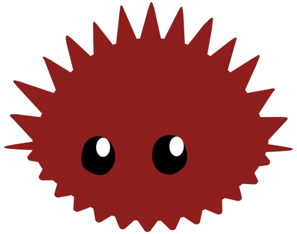


In other words,

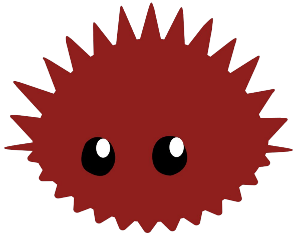
we need a **backend-focused distribution**



> That’s **impossible**!



That’s **exactly** what every single company using Rust ends up building once they scale beyond toy examples



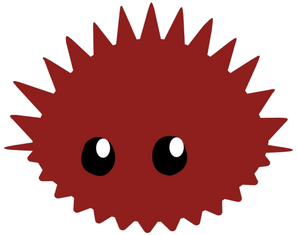
What async executor should we use?

What web framework?

What database driver?

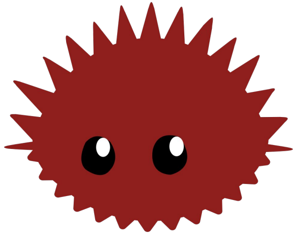
What telemetry libraries?

...



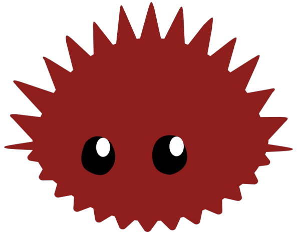
Sometimes it works out,

sometimes it doesn’t

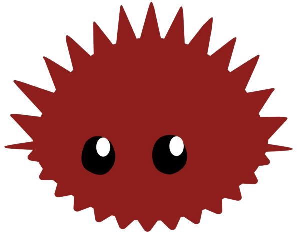


Rust’s weaknesses for backend:

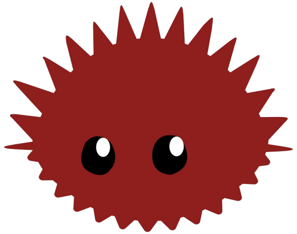
- Limited talent pool with professional experience
- A Lego-like ecosystem
- **A less-than-optimal learning curve**



There’s a **tension** in the Rust ecosystem

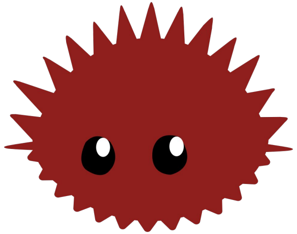


On one side,
we want **great ergonomics**



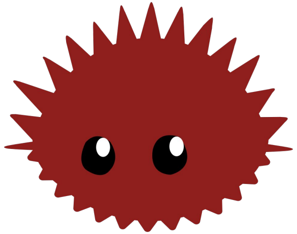
On the other side, we want to

ensure correctness at compile-time

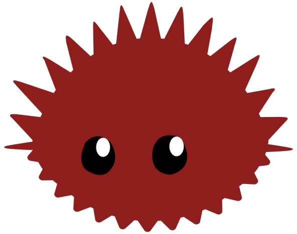


On top of that,

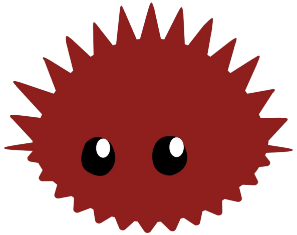
we are building on top of **async Rust**



That’s an **explosive mix**



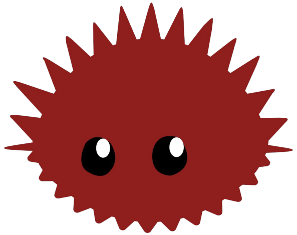
A **beginner has to digest advanced Rust constructs** as soon as they start their **first web project**



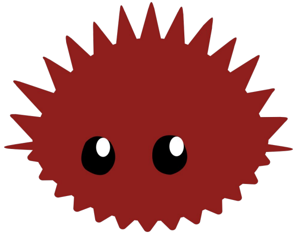
[

*insert screenshot of a compiler error
that fills an entire terminal screen mentioning
traits you’ve never seen before, Send, Sync
and tuples of various lengths*

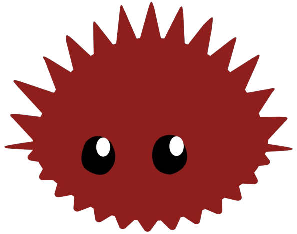
]



That’s a recipe for **churn**

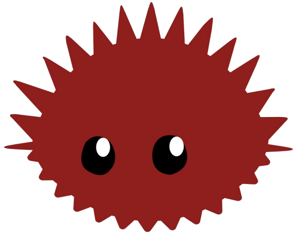


An **experienced mentor** can mitigate these issues, but that’s **a luxury** that few have available



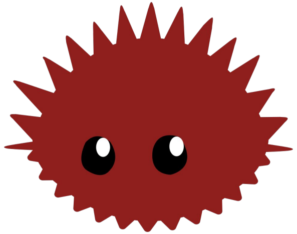
That’s why

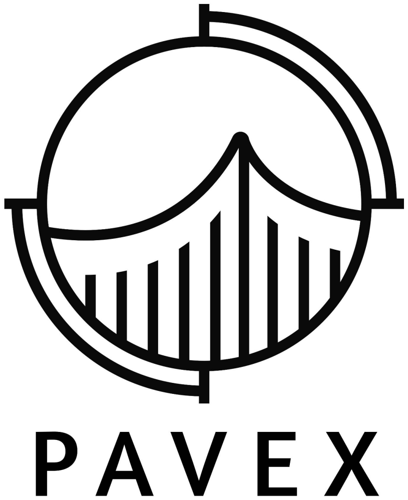
Rust is not a mainstream language for backend development



But it could be!

And I want it to be!




PAVEX

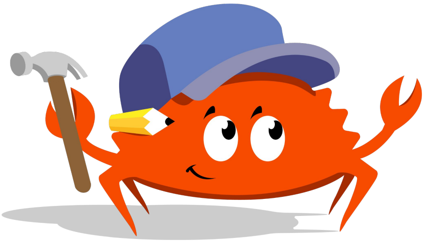
Pavex is a new framework for building Rust APIs

It was born as an experiment,
at the **end of 2022**

Can we offer **a better DX**,

if we choose **a radically different approach**?

Show, don’t tell: **demo time**!



You have seen some of

Pavex **core tenets** in action

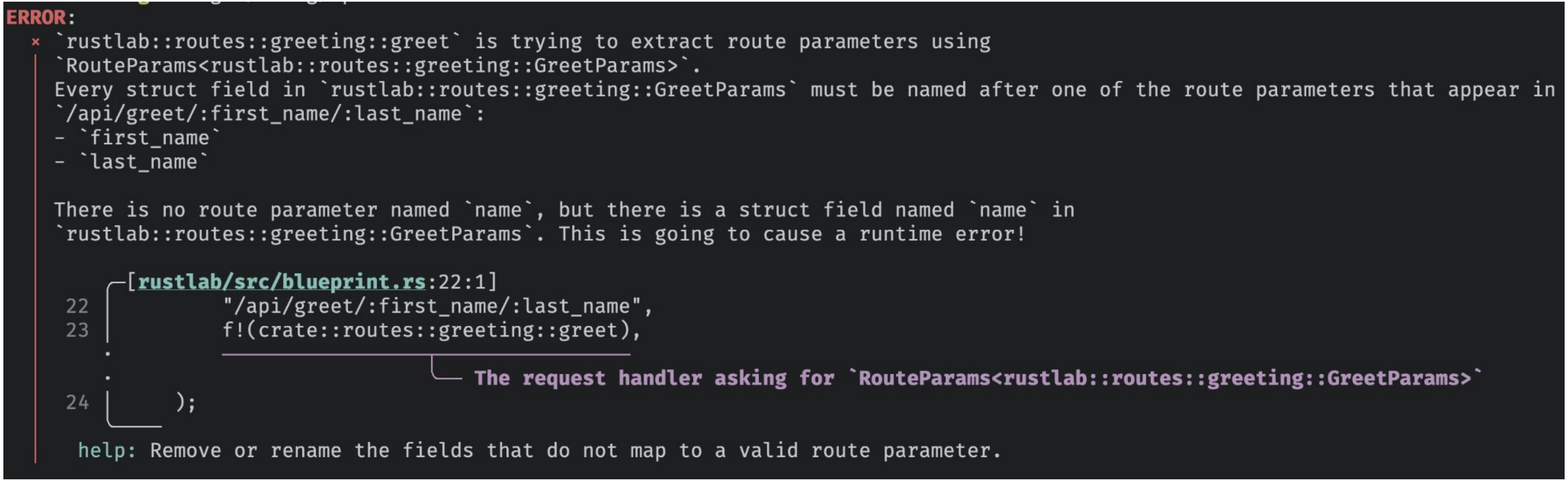
PAVEX

1

High-quality **error messages**

that **speak the language of backend development**

PAVEX



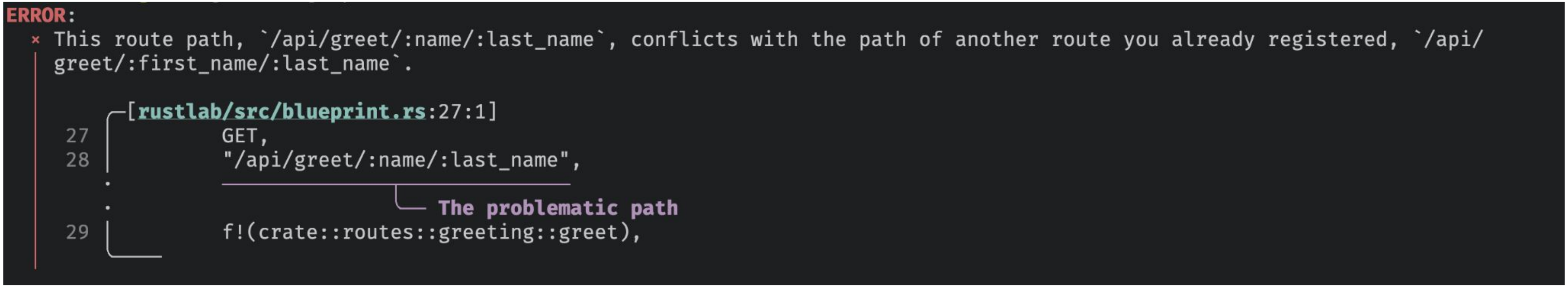
```
ERROR:
  × `rustlab::routes::greeting::greet` is trying to extract route parameters using
  │ `RouteParams<rustlab::routes::greeting::GreetParams>`.
  │ Every struct field in `rustlab::routes::greeting::GreetParams` must be named after one of the route parameters that appear in
  │ `/api/greet/:first_name/:last_name`:
  │ - `first_name`
  │ - `last_name`
  │
  │ There is no route parameter named `name`, but there is a struct field named `name` in
  │ `rustlab::routes::greeting::GreetParams`. This is going to cause a runtime error!
  │
  │     ╭─[rustlab/src/blueprint.rs:22:1]
  │  22 │         "/api/greet/:first_name/:last_name",
  │  23 │         f!(crate::routes::greeting::greet),
  │     ·         ──────────────────────────────────
  │     ·                          ╰── The request handler asking for `RouteParams<rustlab::routes::greeting::GreetParams>`
  │  24 │     );
  │     ╰────
  │   help: Remove or rename the fields that do not map to a valid route parameter.
```

PAVEX

2️⃣

Errors must be **caught at compile-time** where possible

PAVEX



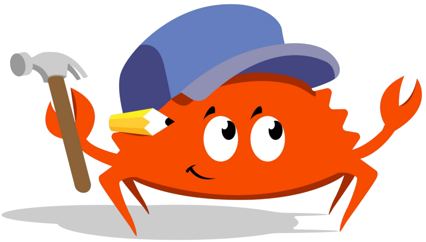
```
ERROR:
  × This route path, `/api/greet/:name/:last_name`, conflicts with the path of another route you already registered, `/api/
  │ greet/:first_name/:last_name`.
  │
  │      ╭─[rustlab/src/blueprint.rs:27:1]
  │   27 │            GET,
  │   28 │            "/api/greet/:name/:last_name",
  │      ·            ─────────────┬─────────────
  │      ·                         ╰── The problematic path
  │   29 │            f!(crate::routes::greeting::greet),
  │      ╰────
```

PAVEX

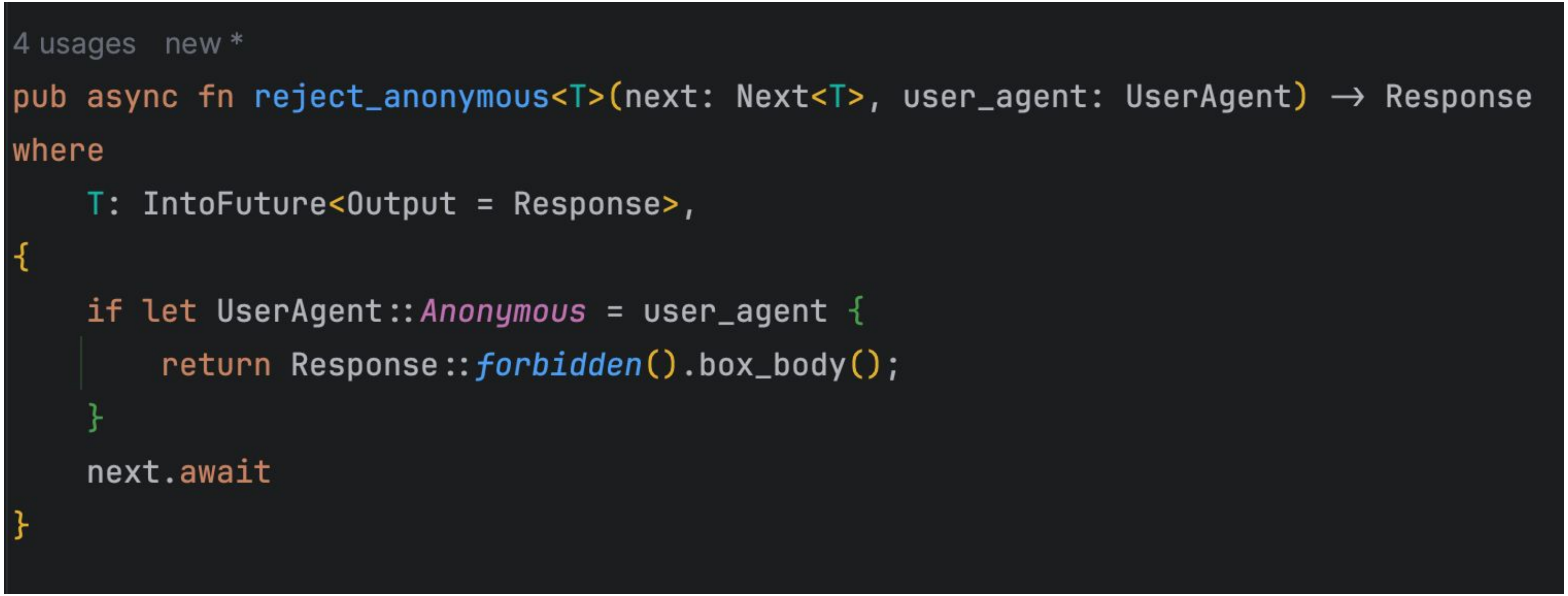
3️⃣

Boring Rust is enough

for the vast majority of tasks



PAVEX



```
4 usages   new *
pub async fn reject_anonymous<T>(next: Next<T>, user_agent: UserAgent) -> Response
where
    T: IntoFuture<Output = Response>,
{
    if let UserAgent::Anonymous = user_agent {
        return Response::forbidden().box_body();
    }
    next.await
}
```

PAVEX

4️⃣

Pavex’s problem domain is **building APIs**

PAVEX

It is not limited to understanding the HTTP protocol, routing requests or managing state

PAVEX

It is **all those things**, and **more**:

auth, configuration, testing, client-generation, etc.

PAVEX

We want to look at the end-to-end process
to make it easier to build **high-quality applications**

PAVEX

We won’t get there overnight:

we are **starting from the foundations**

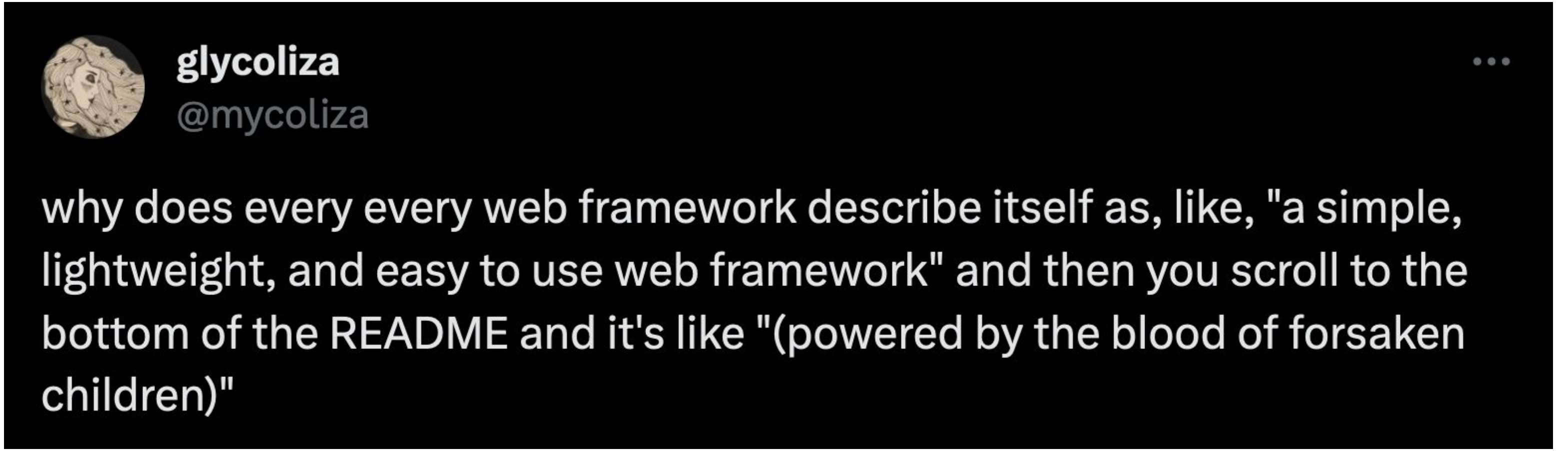
that’ll make it possible

PAVEX

A look under the hood

How does Pavex **actually** work?

PAVEX



glycoliza
@mycoliza

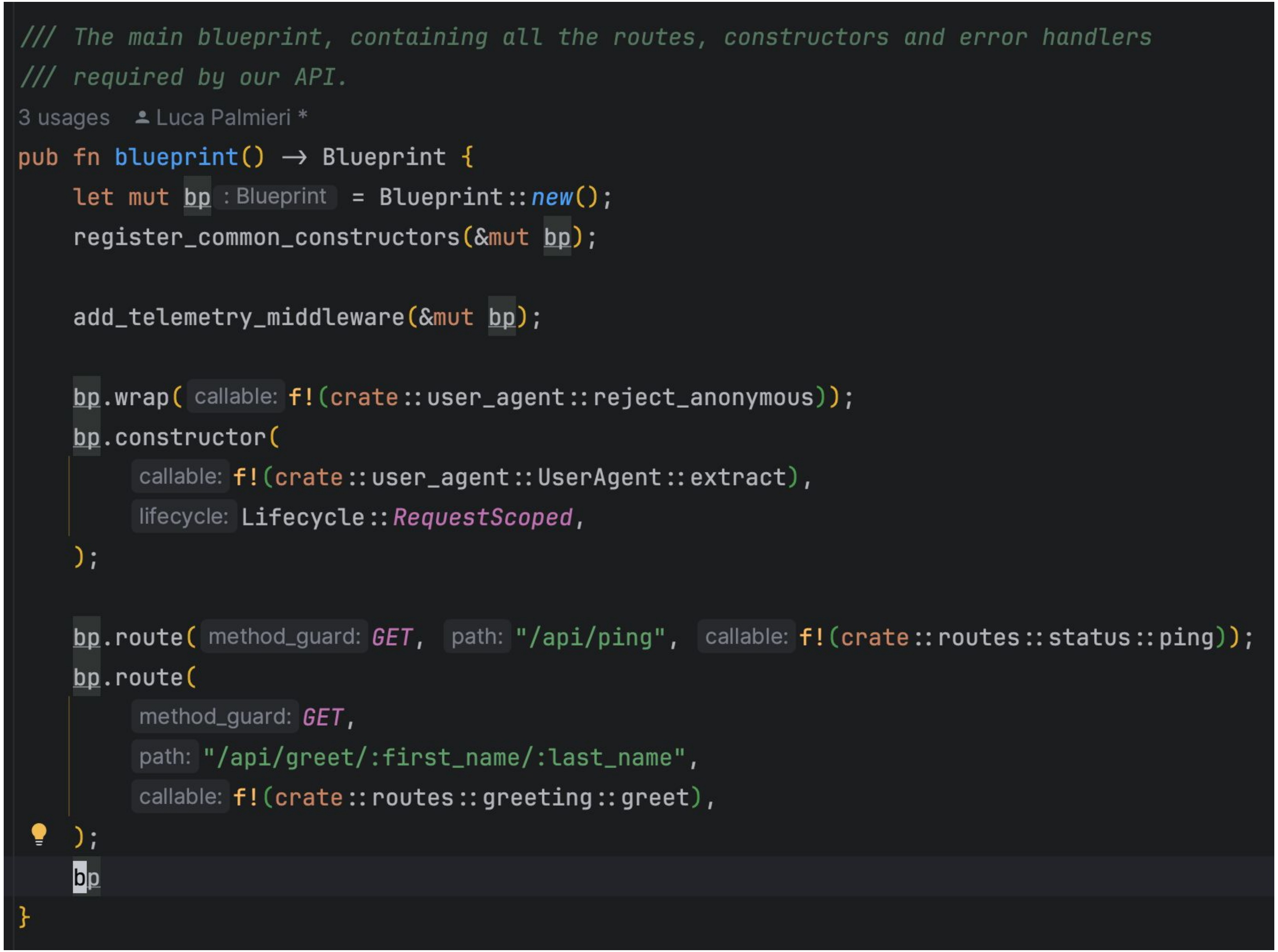
why does every every web framework describe itself as, like, "a simple, lightweight, and easy to use web framework" and then you scroll to the bottom of the README and it's like "(powered by the blood of forsaken children)"

PAVEX

You fill out a

declarative `Blueprint`

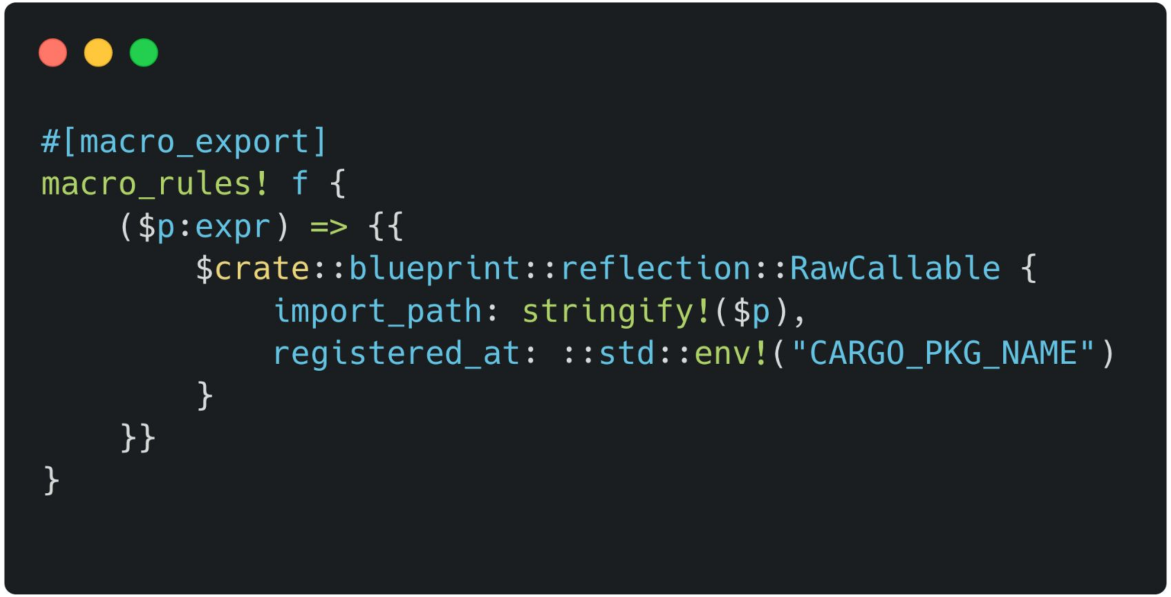
for your application



```
/// The main blueprint, containing all the routes, constructors and error handlers
/// required by our API.
3 usages   Luca Palmieri *
pub fn blueprint() -> Blueprint {
    let mut bp : Blueprint = Blueprint::new();
    register_common_constructors(&mut bp);

    add_telemetry_middleware(&mut bp);

    bp.wrap( callable: f!(crate::user_agent::reject_anonymous));
    bp.constructor(
        callable: f!(crate::user_agent::UserAgent::extract),
        lifecycle: Lifecycle::RequestScoped,
    );

    bp.route( method_guard: GET,  path: "/api/ping",  callable: f!(crate::routes::status::ping));
    bp.route(
        method_guard: GET,
        path: "/api/greet/:first_name/:last_name",
        callable: f!(crate::routes::greeting::greet),
    );
    bp
}
```

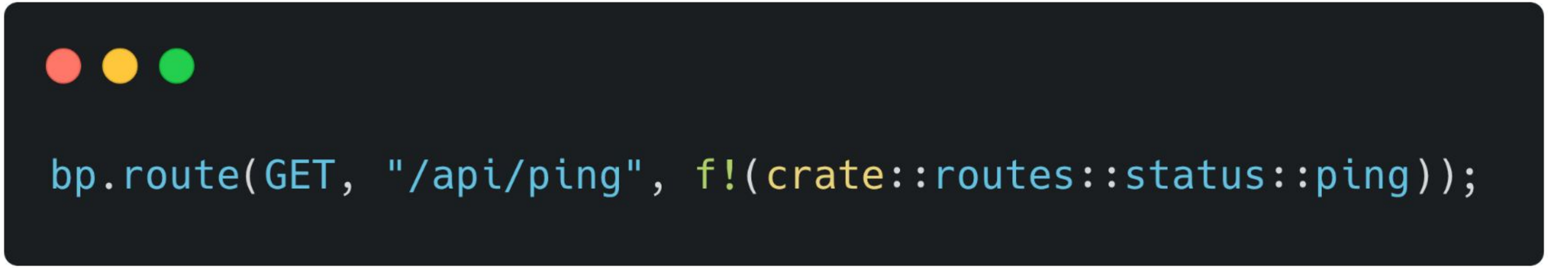
What is that `f!` macro doing?

PAVEX



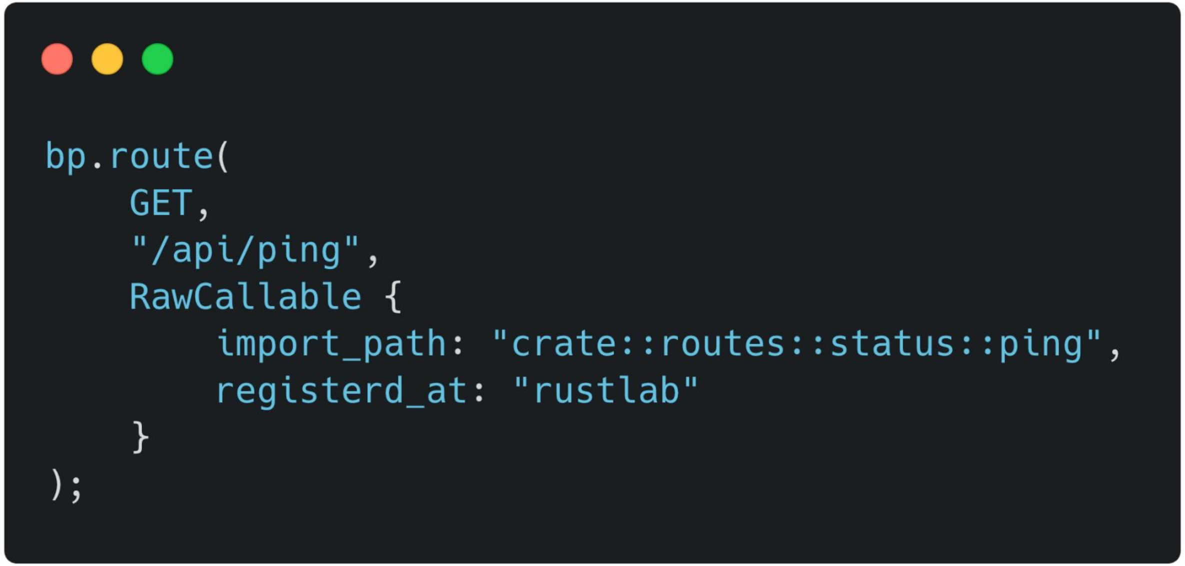
```
#[macro_export]
macro_rules! f {
    ($p:expr) => {{
        $crate::blueprint::reflection::RawCallable {
            import_path: stringify!($p),
            registered_at: ::std::env!("CARGO_PKG_NAME")
        }
    }}
}
```

PAVEX



```
bp.route(GET, "/api/ping", f!(crate::routes::status::ping));
```

PAVEX



```
bp.route(
    GET,
    "/api/ping",
    RawCallable {
        import_path: "crate::routes::status::ping",
        registerd_at: "rustlab"
    }
);
```

PAVEX

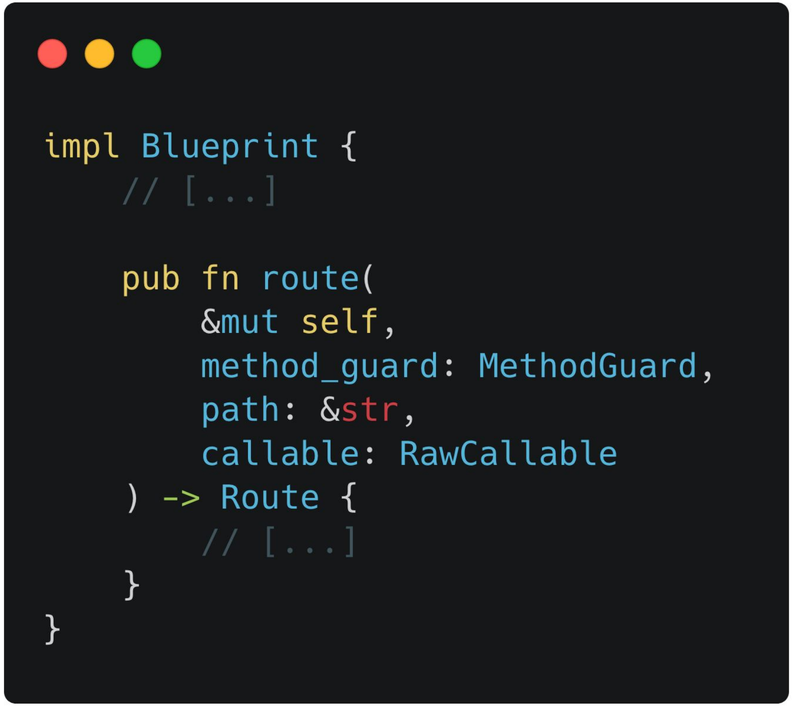
Remember our tenets:

we want **high-quality error messages**

that **speak the language of backend development**

We **don’t rely on trait bounds** for compile-time static analysis

PAVEX



```
impl Blueprint {
    // [...]

    pub fn route(
        &mut self,
        method_guard: MethodGuard,
        path: &str,
        callable: RawCallable
    ) -> Route {
        // [...]
    }
}
```

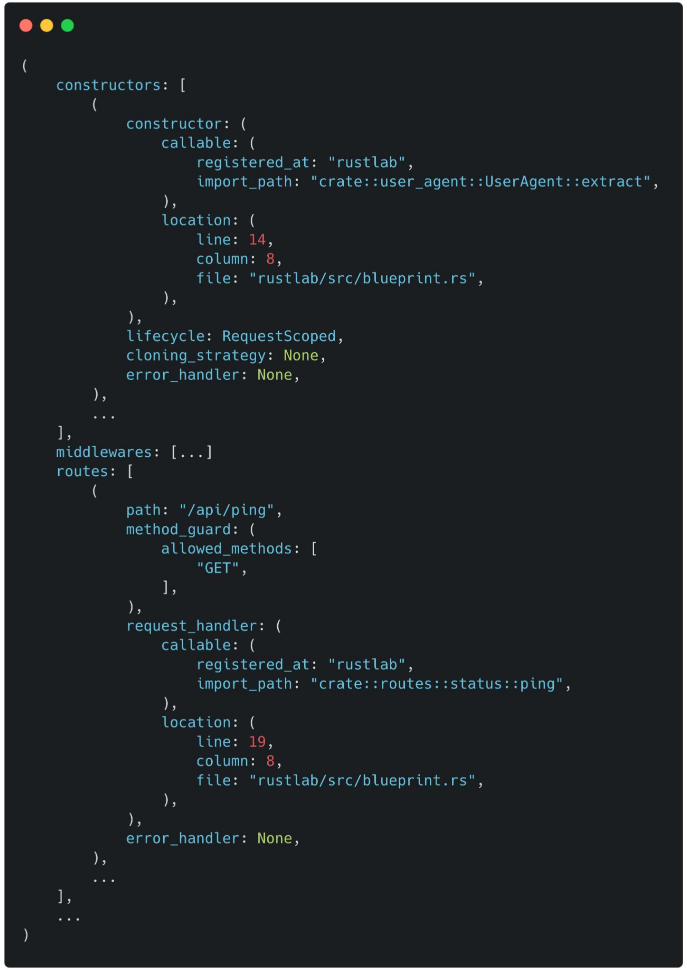
PAVEX

Validation and analysis are **deferred** to **Pavex's transpiler**

```
pavex generate [...]
```

PAVEX

The `Blueprint` is **serialized** and passed to **Pavex's transpiler** as input



```
(
    constructors: [
        (
            constructor: (
                callable: (
                    registered_at: "rustlab",
                    import_path: "crate::user_agent::UserAgent::extract",
                ),
                location: (
                    line: 14,
                    column: 8,
                    file: "rustlab/src/blueprint.rs",
                ),
            ),
            lifecycle: RequestScoped,
            cloning_strategy: None,
            error_handler: None,
        ),
        ...
    ],
    middlewares: [...]
    routes: [
        (
            path: "/api/ping",
            method_guard: (
                allowed_methods: [
                    "GET",
                ],
            ),
            request_handler: (
                callable: (
                    registered_at: "rustlab",
                    import_path: "crate::routes::status::ping",
                ),
                location: (
                    line: 19,
                    column: 8,
                    file: "rustlab/src/blueprint.rs",
                ),
            ),
            error_handler: None,
        ),
        ...
    ],
    ...
)
```

The **transpiler** is where

all the **compile-time validation** takes place

If there are no errors,
the transpiler… transpiles!

It **generates a new a crate**

from your `Blueprint`:

the server SDK

The code in **the server SDK combines everything together**: request handlers, constructors and middlewares

PAVEX

Let’s explore the generated code
to get an understanding of what it entails

PAVEX

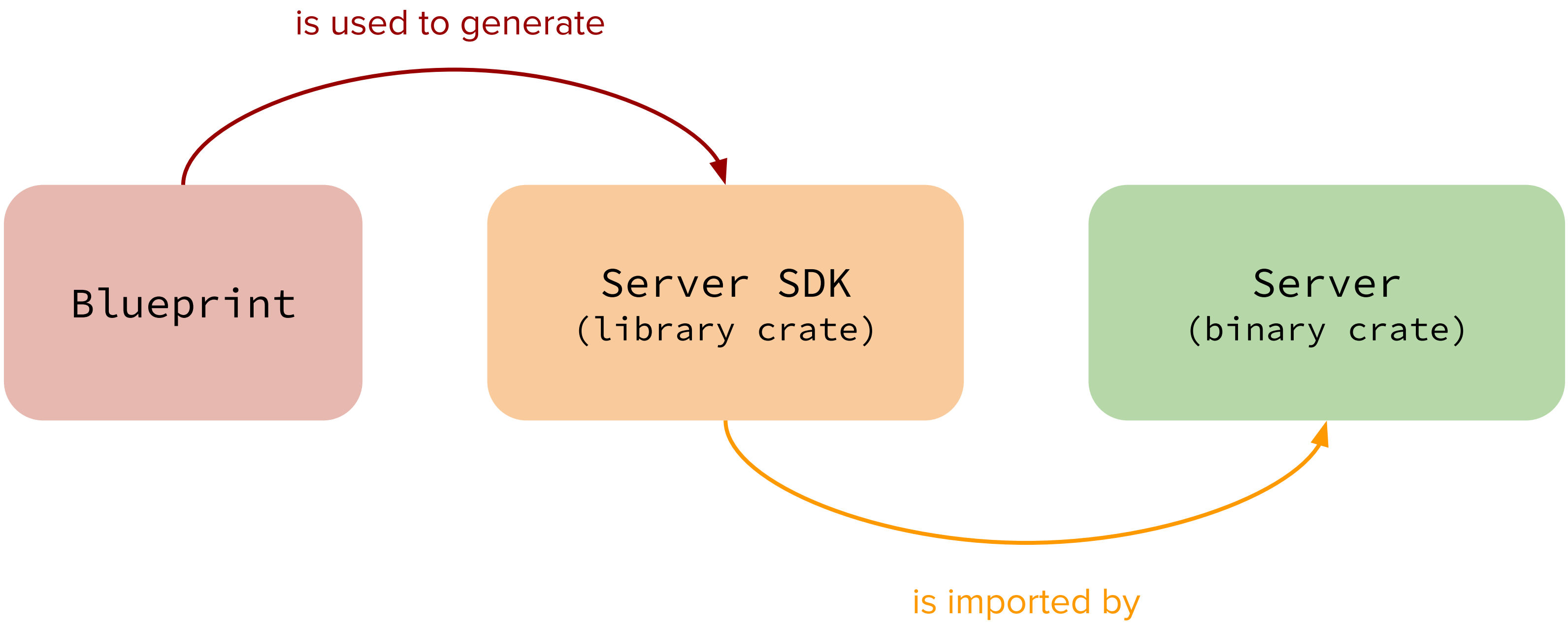
At the top level,

the server SDK exposes two key items:

`run` and **`ApplicationState`**

PAVEX

With those two items,
you can assemble the **server binary**,
the executable that will serve incoming requests



is used to generate

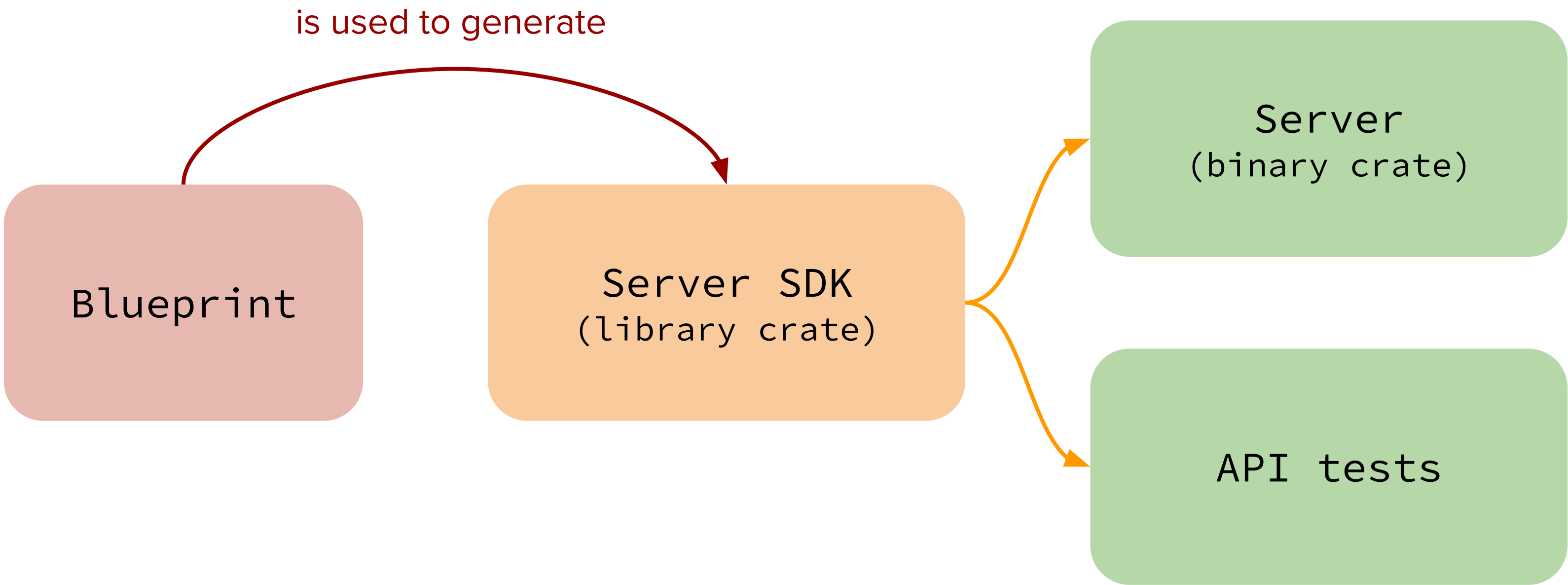
Blueprint

Server SDK
(library crate)

Server
(binary crate)

is imported by

PAVEX



is used to generate

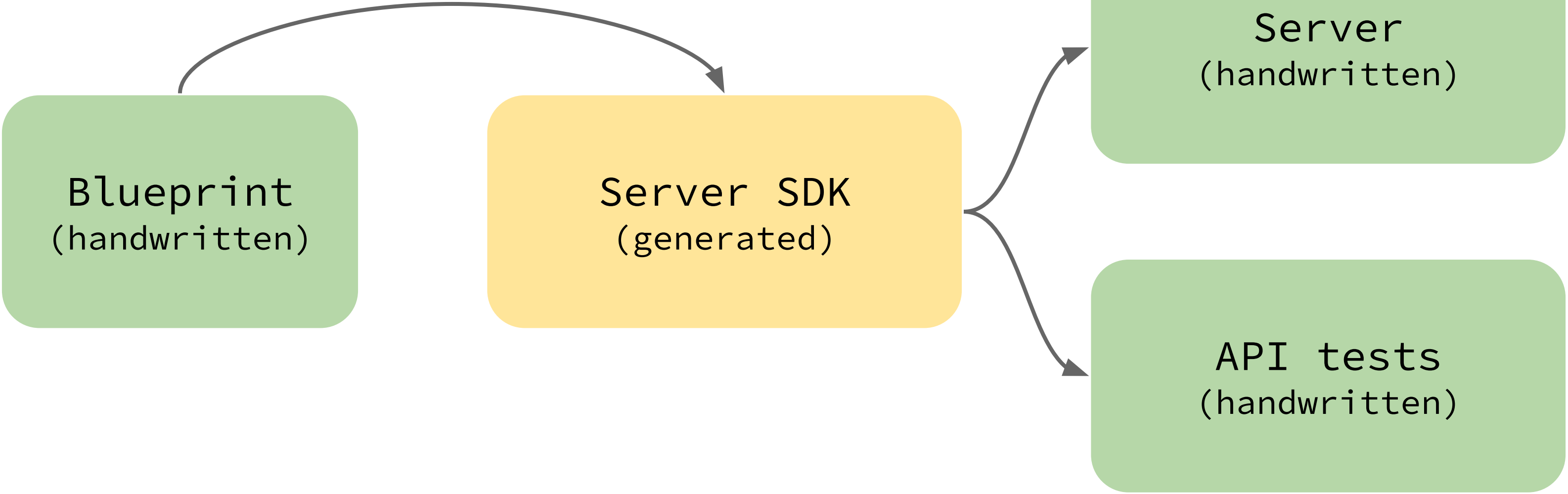
Blueprint

Server SDK
(library crate)

Server
(binary crate)

API tests

PAVEX



Blueprint
(handwritten)

Server SDK
(generated)

Server
(handwritten)

API tests
(handwritten)

PAVEX

Why do we need three crates?

Why don’t we *just* use a macro, or a build script?

Pavex's secret sauce is

a compile-time reflection engine

What inputs does this request handler take?

What output does it return?

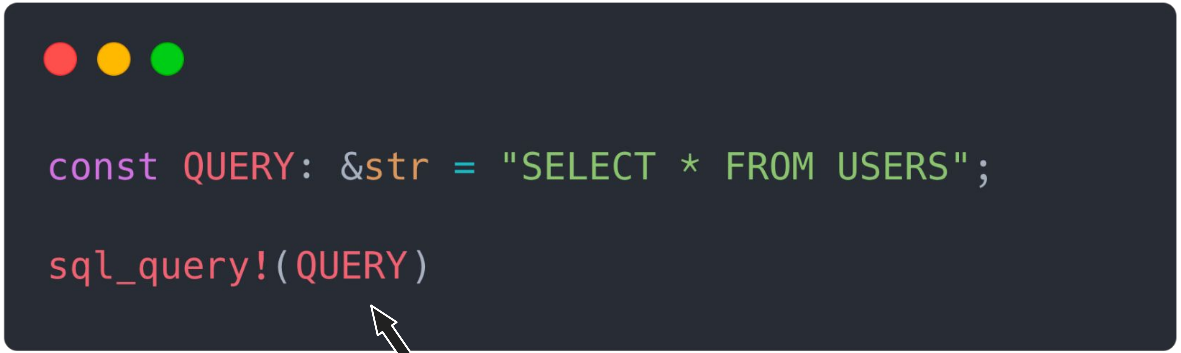
Do we have a constructor registered for this type?

…

PAVEX

We want to **answer those questions**,
and we want to do it **at compile-time**

Macros in Rust **operate on tokens,**
they have **no access to type-level information**



```
const QUERY: &str = "SELECT * FROM USERS";

sql_query!(QUERY)
```

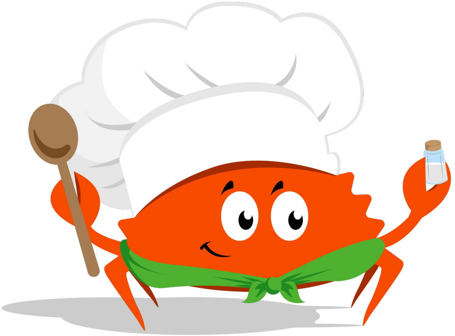
The macro can’t resolve this!

PAVEX

Macros won’t cut it,
what can we use?

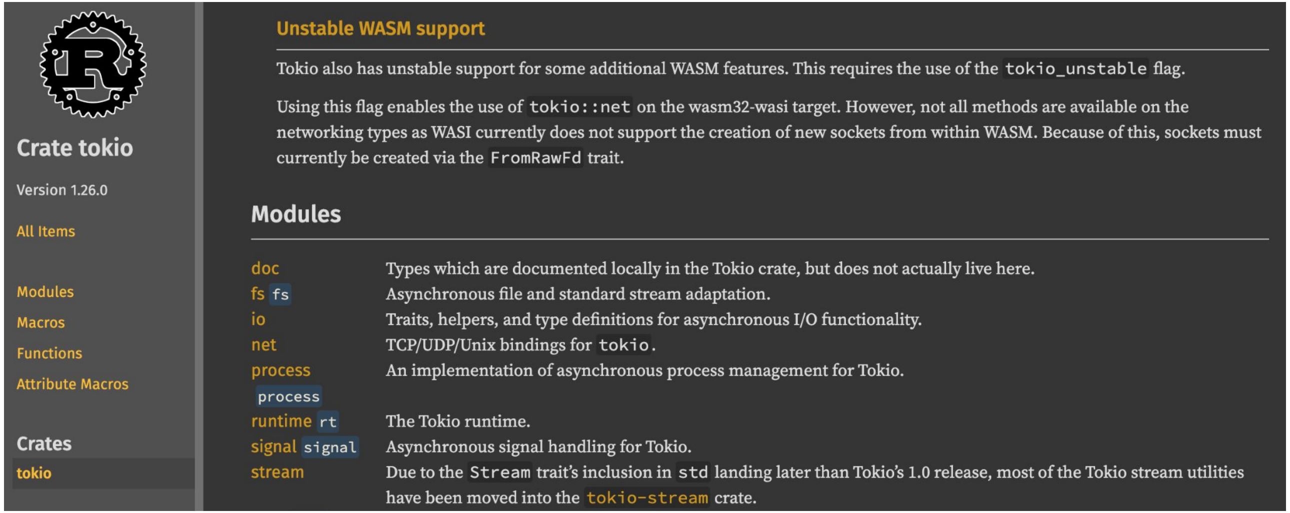
PAVEX

The reflection engine



Pavex is powered by **rustdoc-json**

Where does your mind go
when I say **rustdoc**?



Crate tokio

Version 1.26.0

All Items

Modules

Macros

Functions

Attribute Macros

Crates

tokio

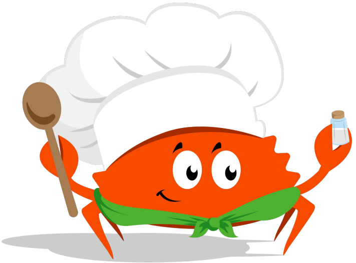
Unstable WASM support

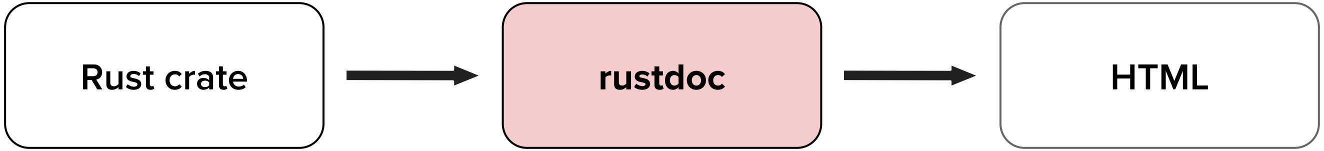
Tokio also has unstable support for some additional WASM features. This requires the use of the `tokio_unstable` flag.

Using this flag enables the use of `tokio::net` on the wasm32-wasi target. However, not all methods are available on the networking types as WASI currently does not support the creation of new sockets from within WASM. Because of this, sockets must currently be created via the `FromRawFd` trait.

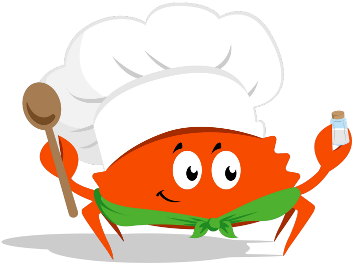
Modules

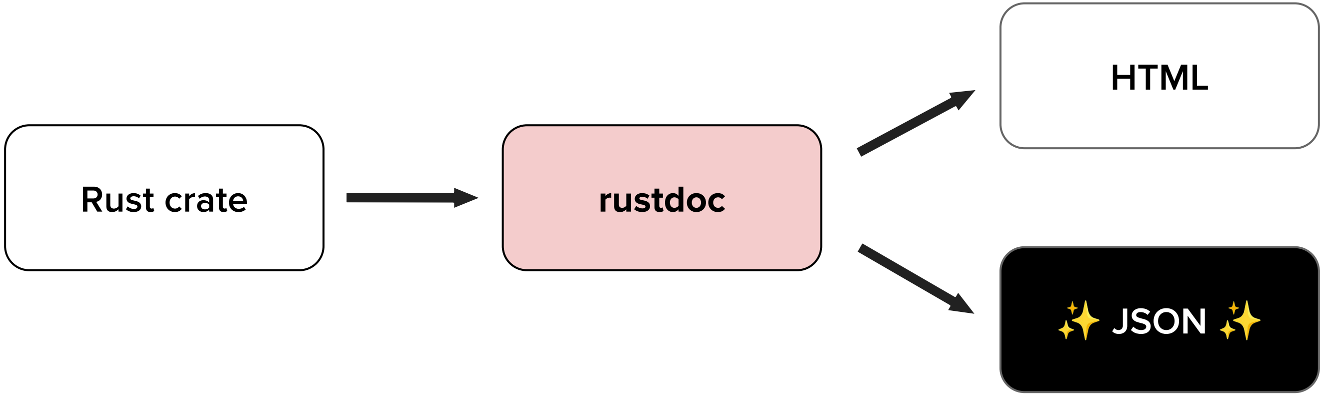
doc	Types which are documented locally in the Tokio crate, but does not actually live here.
fs `fs`	Asynchronous file and standard stream adaptation.
io	Traits, helpers, and type definitions for asynchronous I/O functionality.
net	TCP/UDP/Unix bindings for `tokio`.
process `process`	An implementation of asynchronous process management for Tokio.
runtime `rt`	The Tokio runtime.
signal `signal`	Asynchronous signal handling for Tokio.
stream	Due to the `Stream` trait's inclusion in `std` landing later than Tokio's 1.0 release, most of the Tokio stream utilities have been moved into the `tokio-stream` crate.





Rust crate → rustdoc → HTML



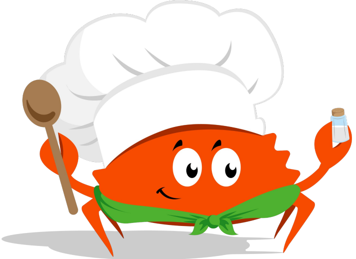


Rust crate

rustdoc

HTML

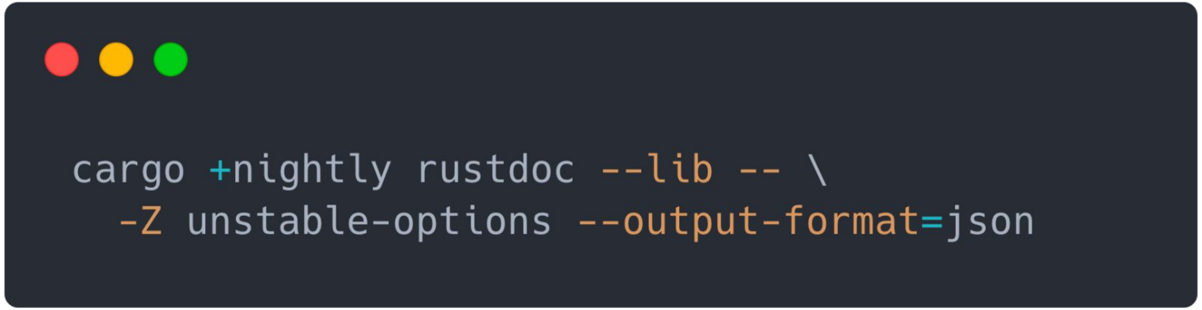
✨ JSON ✨



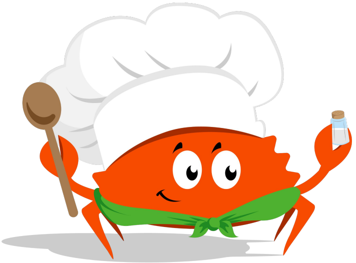
Same information as docs.rs,
in a **machine-parsable format**!

Currently on **nightly**,

introduced in an RFC from June 2020

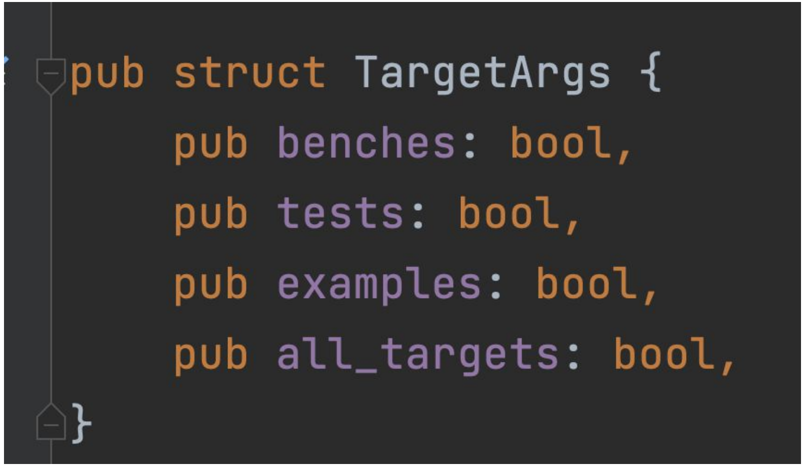


```
cargo +nightly rustdoc --lib -- \
  -Z unstable-options --output-format=json
```

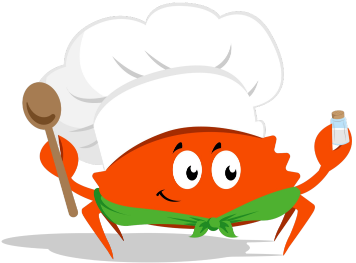


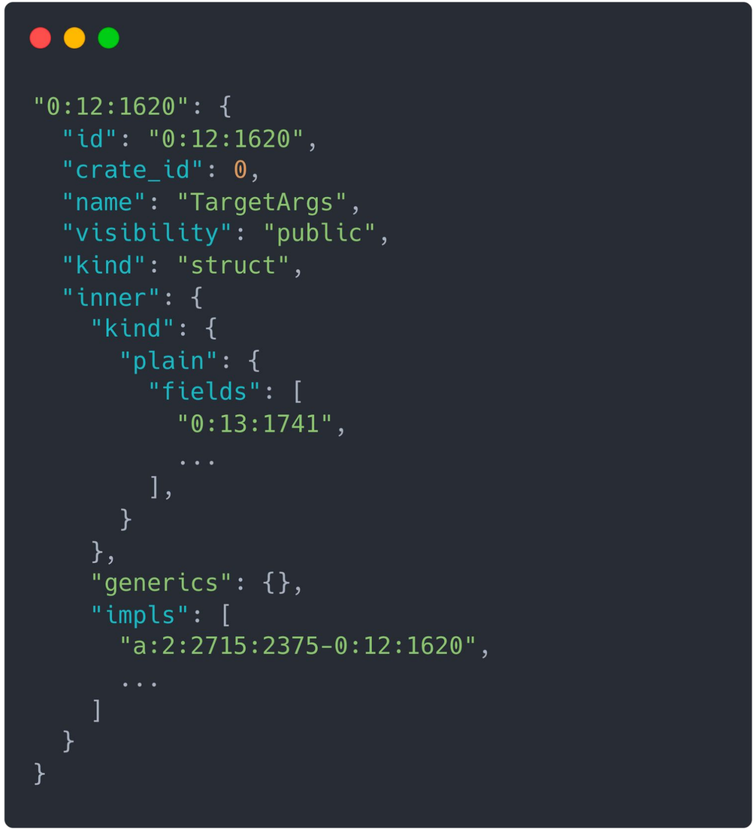
Let’s look at an **example**:

a **struct** from cargo-chef



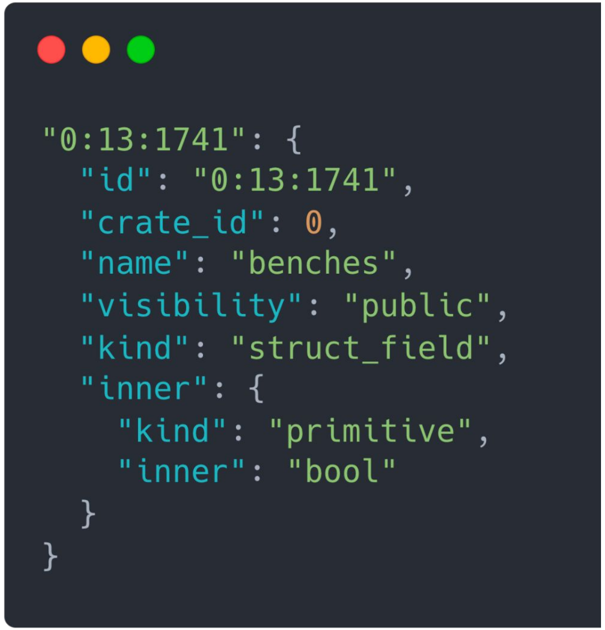
```rust
pub struct TargetArgs {
    pub benches: bool,
    pub tests: bool,
    pub examples: bool,
    pub all_targets: bool,
}
```



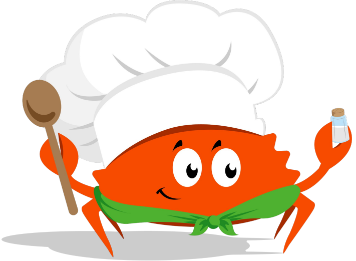


```
"0:12:1620": {
  "id": "0:12:1620",
  "crate_id": 0,
  "name": "TargetArgs",
  "visibility": "public",
  "kind": "struct",
  "inner": {
    "kind": {
      "plain": {
        "fields": [
          "0:13:1741",
          ...
        ],
      }
    },
    "generics": {},
    "impls": [
      "a:2:2715:2375-0:12:1620",
      ...
    ]
  }
}
```

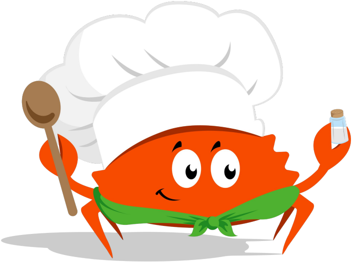
If you follow the ids…

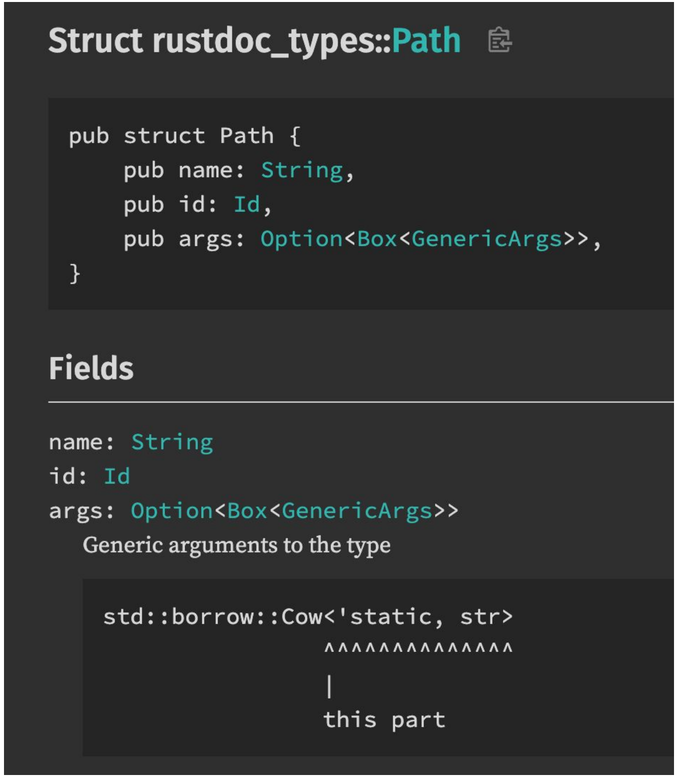


```
"0:13:1741": {
  "id": "0:13:1741",
  "crate_id": 0,
  "name": "benches",
  "visibility": "public",
  "kind": "struct_field",
  "inner": {
    "kind": "primitive",
    "inner": "bool"
  }
}
```



You can use **rustdoc-types** to parse the raw JSON into Rust structs





Struct rustdoc_types::Path

```
pub struct Path {
    pub name: String,
    pub id: Id,
    pub args: Option<Box<GenericArgs>>,
}
```

Fields

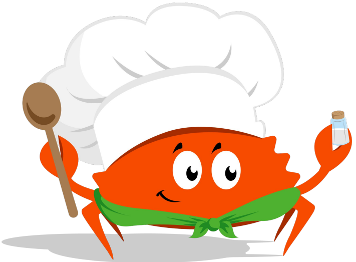
`name: String`

`id: Id`

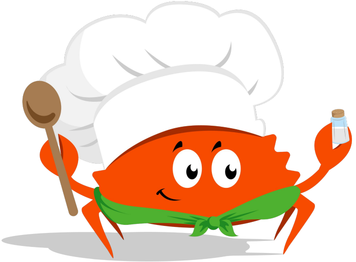
`args: Option<Box<GenericArgs>>`

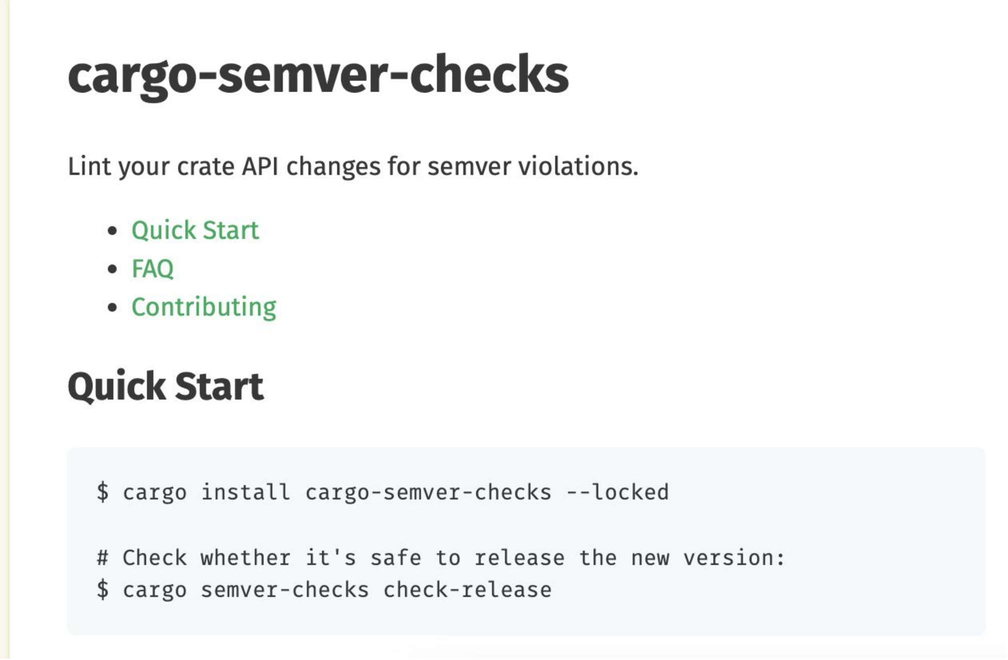
Generic arguments to the type

```
std::borrow::Cow<'static, str>
                ^^^^^^^^^^^^^^
                |
                this part
```



Rustdoc’s JSON format **is enabling**
a new generation of Rust tooling





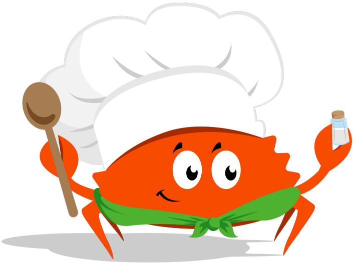
cargo-semver-checks

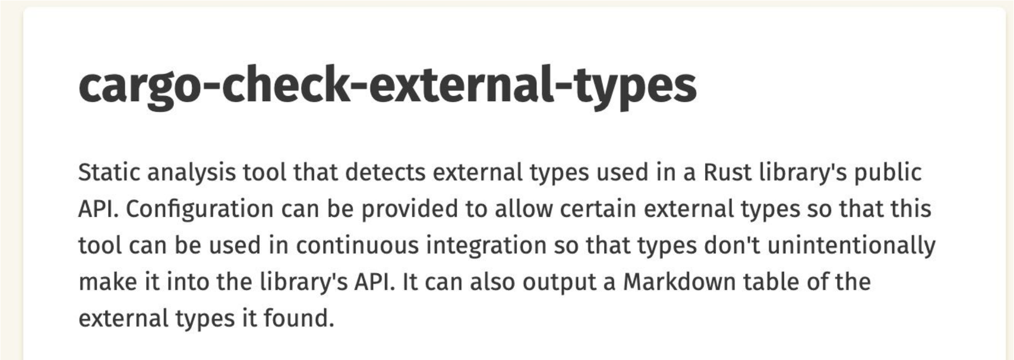
Lint your crate API changes for semver violations.

- Quick Start
- FAQ
- Contributing

Quick Start

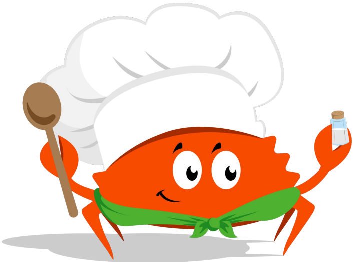
```
$ cargo install cargo-semver-checks --locked

# Check whether it's safe to release the new version:
$ cargo semver-checks check-release
```

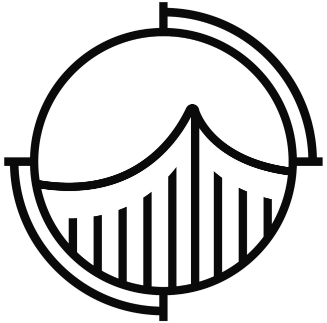




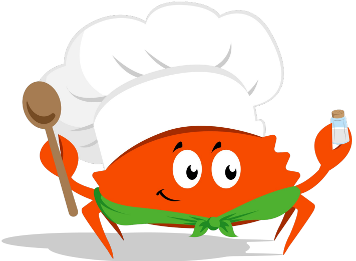
cargo-check-external-types

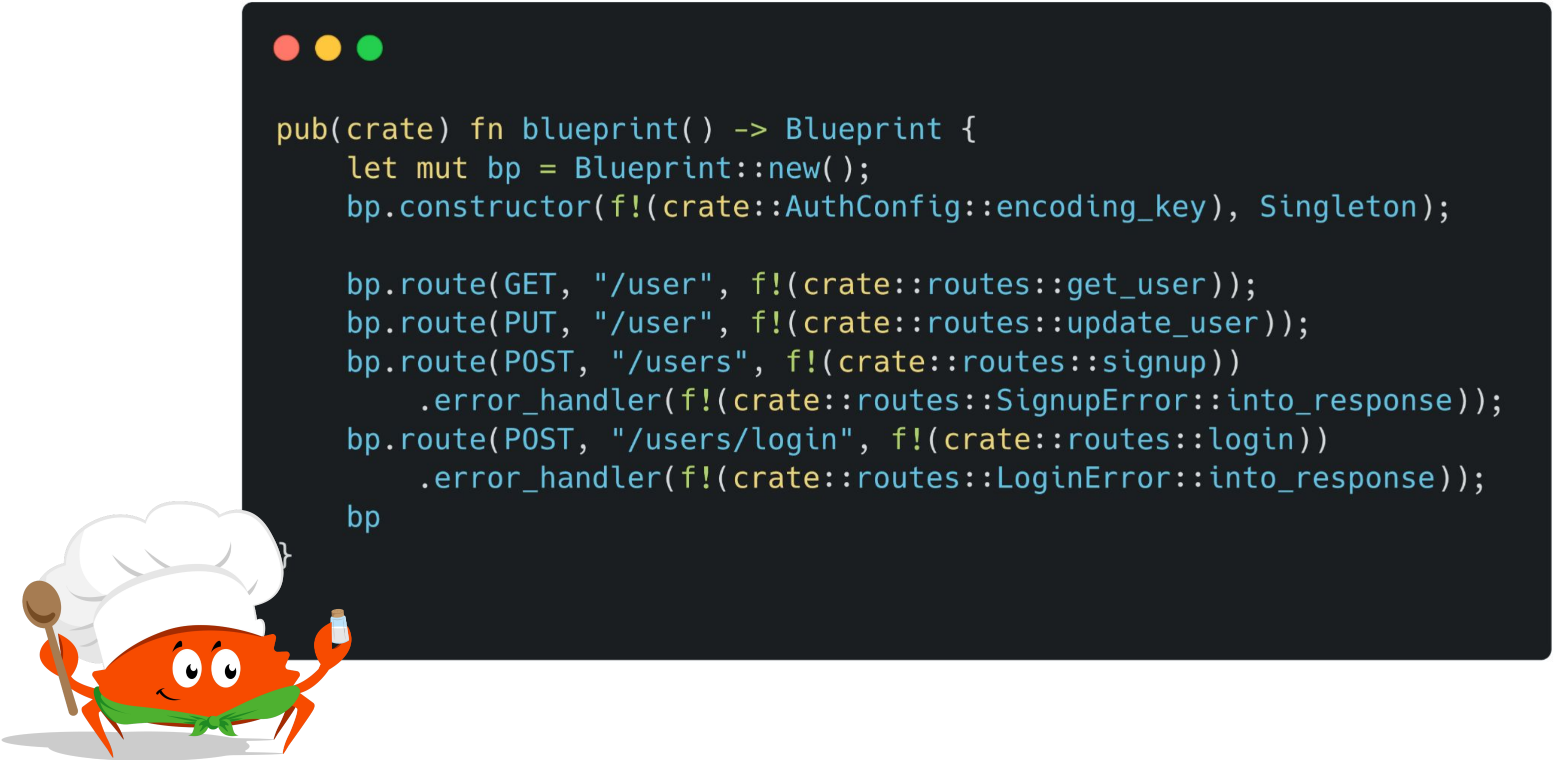
Static analysis tool that detects external types used in a Rust library's public API. Configuration can be provided to allow certain external types so that this tool can be used in continuous integration so that types don't unintentionally make it into the library's API. It can also output a Markdown table of the external types it found.





PAVEX

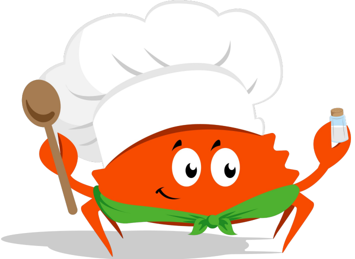




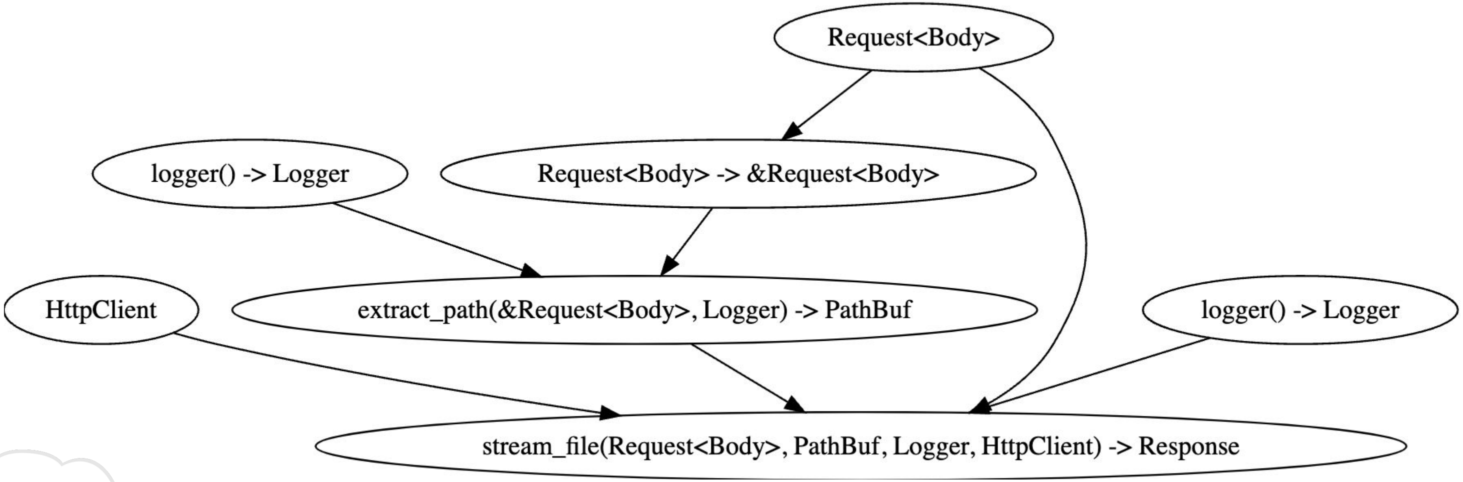
```rust
pub(crate) fn blueprint() -> Blueprint {
    let mut bp = Blueprint::new();
    bp.constructor(f!(crate::AuthConfig::encoding_key), Singleton);

    bp.route(GET, "/user", f!(crate::routes::get_user));
    bp.route(PUT, "/user", f!(crate::routes::update_user));
    bp.route(POST, "/users", f!(crate::routes::signup))
        .error_handler(f!(crate::routes::SignupError::into_response));
    bp.route(POST, "/users/login", f!(crate::routes::login))
        .error_handler(f!(crate::routes::LoginError::into_response));
    bp
}
```

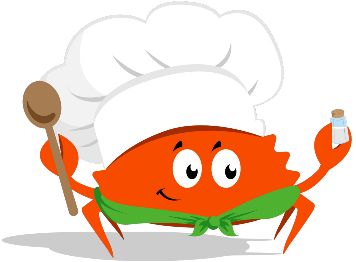
Given a fully qualified path:

- → Determine the crate it was defined into
- → Generate JSON docs for that crate
- → Look up type information

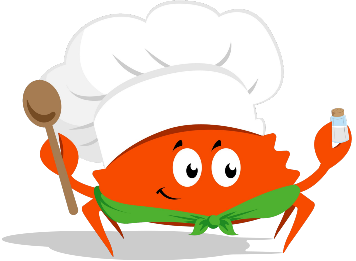


Combining everything together,
we build a **call graph** for each request handler

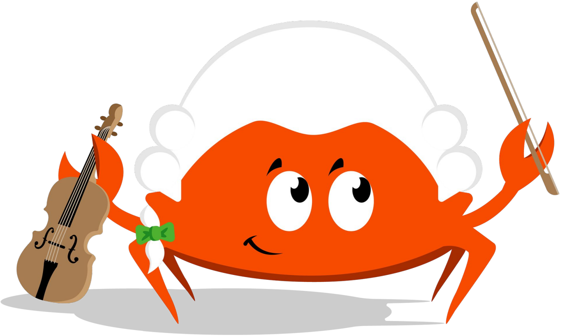

Request<Body>
logger() -> Logger
Request<Body> -> &Request<Body>
HttpClient
extract_path(&Request<Body>, Logger) -> PathBuf
logger() -> Logger
stream_file(Request<Body>, PathBuf, Logger, HttpClient) -> Response



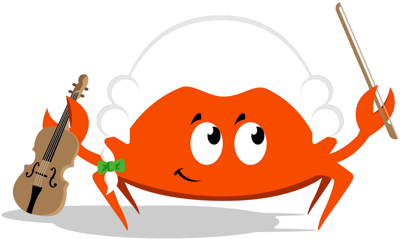
The call graph is used for **static analysis**
and, at the end, to **drive code generation**



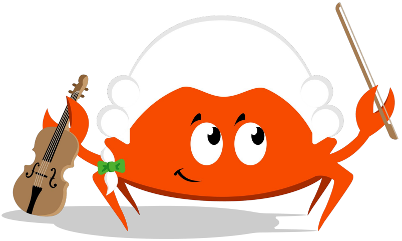
Wrapping up



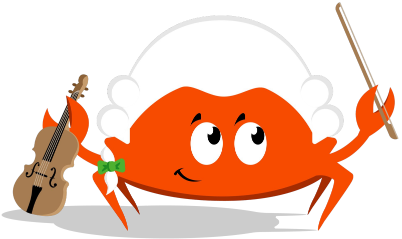
You **can move complexity** around,
but you **cannot eliminate it**



Complexity has to live somewhere

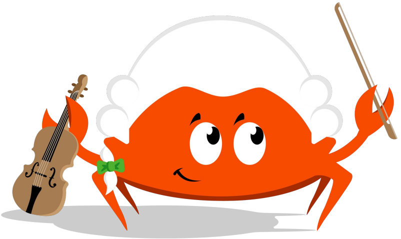


We want Pavex to take on that complexity,
so that **you don't have to**

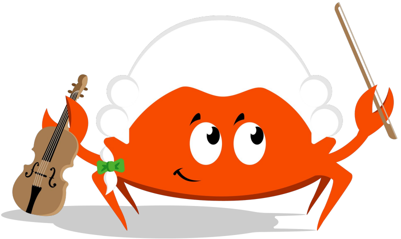


PAVEX

I built a transpiler **because I had to**



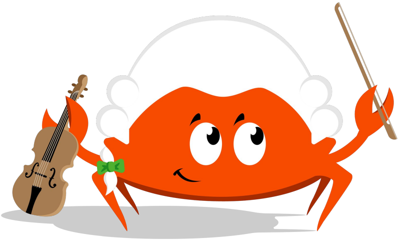
Today we are **just scratching the surface**,
the foundation we'll build on top of



PAVEX

Staged compilation opens up

a universe of possibilities!



PAVEX

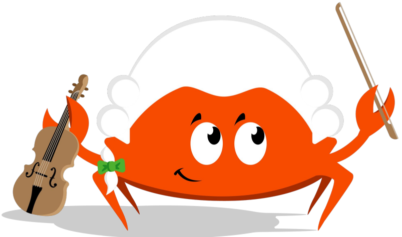
Auto-instrumentation

Accurate OpenAPI specifications

Automatically exploit concurrency opportunities

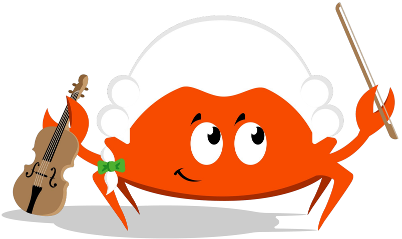
Multiple deployment targets

...



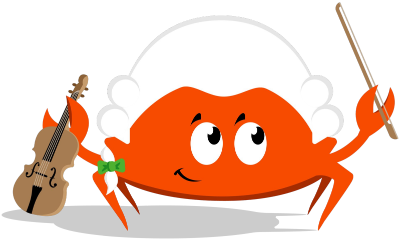
PAVEX

“OK, OK, it’s enough, you convinced me
This Pavex stuff looks super cool,
how do I install it?”



PAVEX

Pavex is **not (yet) generally available**

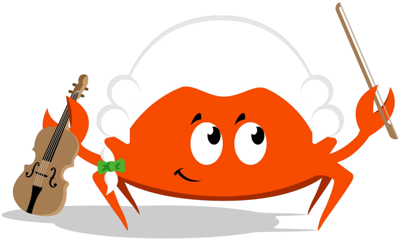


PAVEX

We are going to run a **closed beta**:

you can **join the waiting list** at

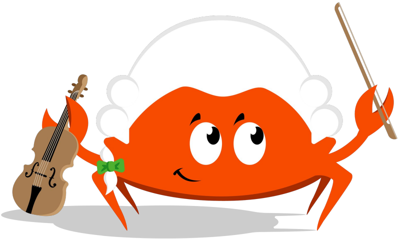
pavex.dev



PAVEX

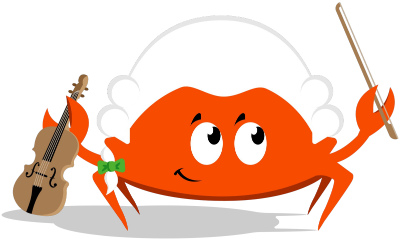
⚠️

Pavex is open source,
but it is a **commercial project**



PAVEX

Feel free to grab me in the hallway or at lunch, happy to discuss further and give demos!



PAVEX

The End

Luca Palmieri

@algo_luca

All the beautiful Ferris illustrations
were created by Esther Arzola

Question time!

Luca Palmieri

@algo_luca

All the beautiful Ferris illustrations
were created by Esther Arzola