Using Rust to develop web applications

An exploration of the Rust ecosystem

Who's this guy?

Sylvain Wallez - @bluxte

Tech lead - Elastic Cloud

Previously tech lead, CTO, architect, trainer, developer…
...at OVH, Actoboard, Sigfox, Scoop.it, Joost, Anyware

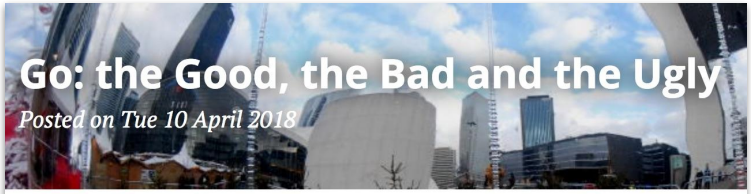
Member of the Apache Software Foundation since 2003

elastic - we're hiring!

On the menu

- Rust for webapps? Why?
- Architecture
- Handling http requests
- Database access
- Logs & metrics
- Docker-ization
- Conclusion

How it all started



This is an additional post in the "Go is not good" series. Go does have some nice features, hence the "The Good" part in this post, but overall I find it cumbersome and painful to use when we go beyond API or network servers (which is what it was designed for) and use it for business domain logic. But even for network programming, it has a lot of gotchas both in its design and implementation that make it dangerous under an apparent simplicity.

What motivated this post is that I recently came back to using Go for a side project. I used Go extensively in my previous job to write a network proxy (both http and raw tcp) for a SaaS service. The network part was rather pleasant (I was also discovering the language), but the accounting and billing part that came with it was painful. As my side project was a simple API I thought using Go would be the right tool to get the job done quickly, but as we know many projects grow beyond their initial scope, so I had to write some data processing to compute statistics and the pains of Go came back. So here's my take on Go woes.

Some background: I love statically typed languages. My first significant programs were written in Pascal. I then used Ada and C/C++ when I started working in the early 90's. I later moved to Java and finally Scala (with some Go in between) and

#3 on HackerNews 💪

"Wait – how does this work in Rust, this "other" recent low-level language?"

"Ooooh, Rust is sweet!"

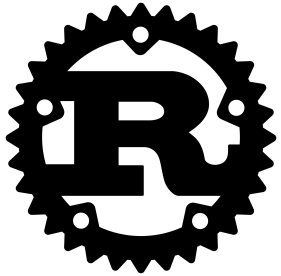
"Can we do webapps with Rust?"

"Yes, we can! And it's quite nice!"

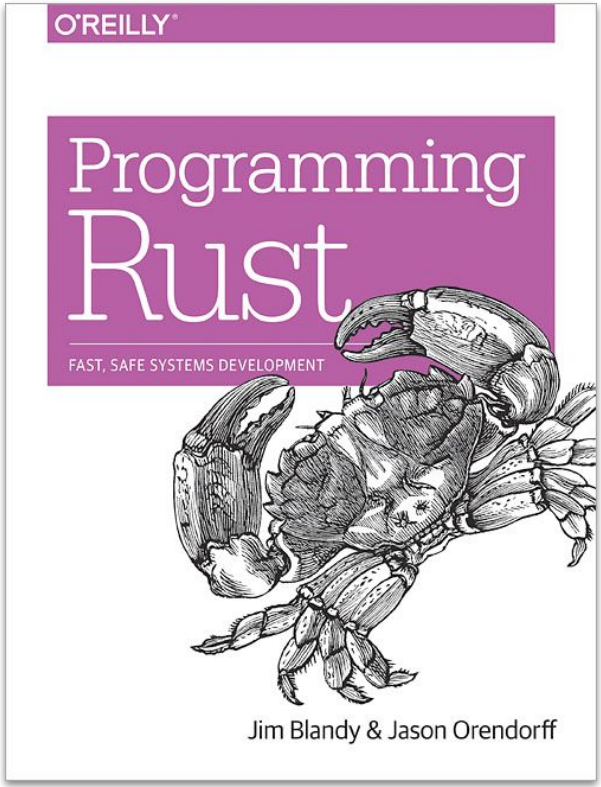
Rust

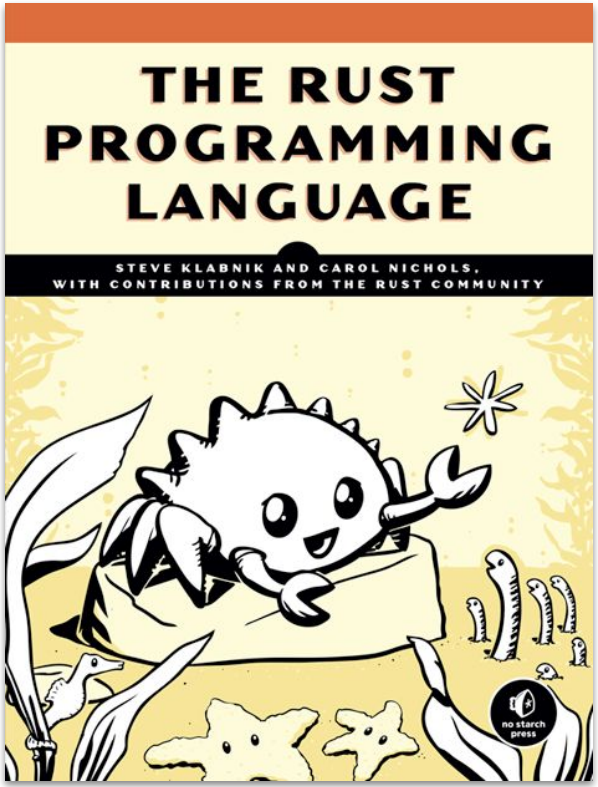
“Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety” – https://www.rust-lang.org/

- Zero cost abstractions, threads without data races, minimal runtime
- No garbage collector, guaranteed memory safety
- Type inference, traits, pattern matching, type classes, higher-order functions



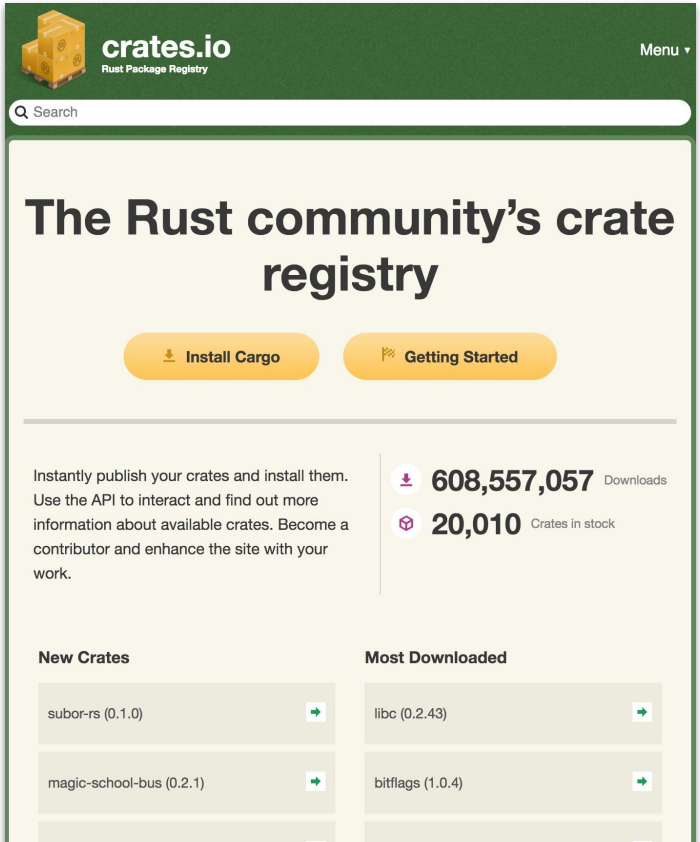
Learning Rust





Also online at https://www.rust-lang.org/

The Rust ecosystem



- crates.io – there's a crate for that!

- Twitter: @rustlang, @ThisWeekInRust
- https://users.rust-lang.org
- https://exercism.io/

- http://www.arewewebyet.org/
- http://arewegameyet.com/
- https://areweideyet.com/
- http://www.arewelearningyet.com/

The project: my blog's comment server

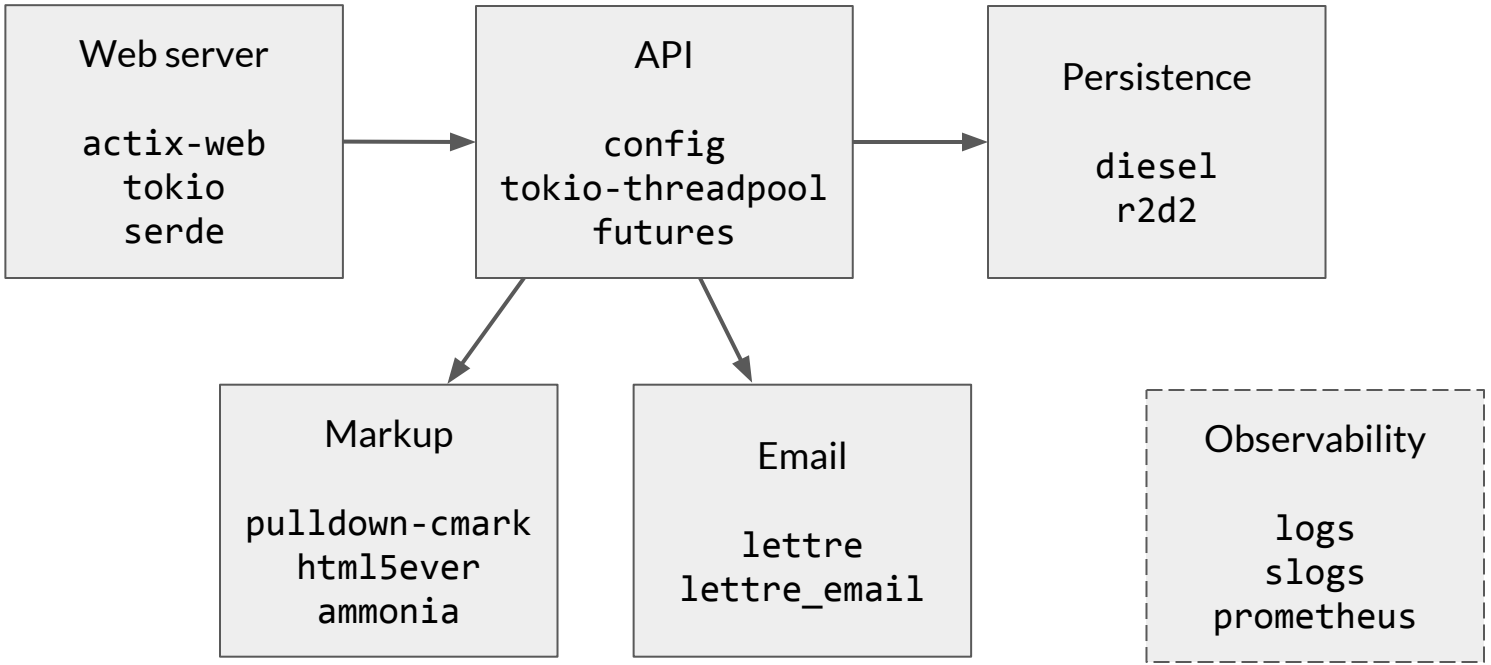
It's in Python, let's rewrite it in Rust

Small, but covers a lot:

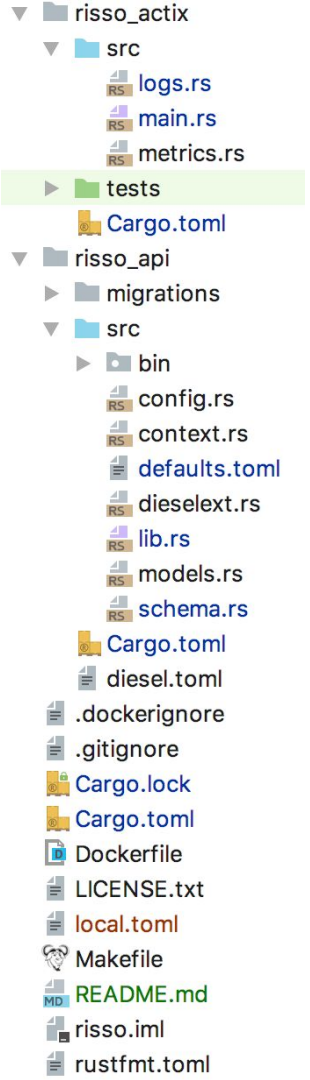
- web api, data validation, CORS
- database access
- markdown, rendering, HTML sanitization
- sending emails
- admin front-end

Code at https://github.com/swallez/risso

Architecture



Project layout



Cargo.toml

```
[package]
name = "risso_actix"
version = "0.1.0"
description = "Actix-web server front-end to risso_api"
authors = ["Sylvain Wallez <sylvain@bluxte.net>"]
license = "Apache-2.0"

[dependencies]
actix = "0.7"
actix-web = "0.7"
actix-web-requestid = "0.1.2"
serde = "1.0.80"
serde_derive = "1.0.80"
failure = "0.1.3"
lazy_static ="1.1"
maplit = "1.0.1"
intern = "0.2.0"
futures = "0.1"
log = "0.4.6"
env_logger = "0.5.13"
slog = "2.4.1"
slog-term = "2.4.0"
slog-json = "2.2.0"
slog-async = "2.3.0"
slog-scope = "4.0.1"
prometheus = "0.4.2"

risso_api = { path = "../risso_api" }
```

main

```rust
pub fn main() -> Result<(), failure::Error> {
   info!("Starting...");

   let config = risso_api::CONFIG.get::<ActixConfig>("actix")?;
   let listen_addr = config.listen_addr;
   let allowed_origins = config.allowed_origins;

   let api_builder = ApiBuilder::new()?;
   let api = api_builder.build();

   let srv = server::new(move || {
       App::with_state(api.clone())
           .route("/", Method::GET, fetch)
           .route("/new", Method::POST, new_comment)
           .route("/id/{id}", Method::GET, view)
           .route("/id/{id}/unsubscribe/{email}/{key}", Method::GET, unsubscribe)
           .route("/metrics", Method::GET, metrics::handler)
            // ...
           .middleware(build_cors(&allowed_origins))
   });

   srv.bind(listen_addr)?.run();
   Ok(())
}
```

config

config.toml

```toml
[actix]
listen_addr = "127.0.0.1:8080"
allowed_origins = []

[database]
db_path = "data/comments.db"
min_connections = 1
max_connections = 10
```

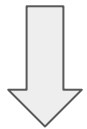

```rust
#[derive(Deserialize)]
pub struct ActixConfig {
    listen_addr: String,
    allowed_origins: Vec<String>,
}

#[derive(Deserialize)]
struct ContextConfig {
    db_path: String,
    min_connections: u32,
    max_connections: u32,
}
```

serde: swiss-army knife of data serialization

Macros and traits that generate key/value (de)constructors.

Libraries providing (de)constructors for specific serialization formats

Any data structure



JSON, CBOR, YAML, MessagePack, TOML, GOB, Pickle, RON, BSON, Avro, URL x-www-form-urlencoded, XML, env vars, AWS Parameter Store and many more...

config

```rust
pub fn load_config() -> Result<Config, ConfigError> {
    let mut s = Config::new();

    // Load defaults
    s.merge(File::from_str(include_str!("defaults.toml"), FileFormat::Toml))?;

    // Find an optional "--config" command-line argument
    let mut args = env::args();
    while let Some(arg) = args.next() {
        if arg == "--config" {
            break;
        }
    }
    if let Some(path) = args.next() {
        s.merge(File::with_name(&path))?;
    }

    // Load an optional local file (useful for development)
    s.merge(File::with_name("local").required(false))?;

    Ok(s)
}
```

ACTIX

rust's powerful actor system and most fun web framework

Type Safe

Forget about stringly typed objects, from request to response, everything has types.

Feature Rich

Actix provides a lot of features out of box. WebSockets, HTTP/2, pipelining etc.

Extensible

Easily create your own libraries that any Actix application can use.

Blazingly Fast

Actix is blazingly fast. Don't take our word for it -- see for yourself!

```rust
extern crate actix_web;
use actix_web::{server, App, HttpRequest, Responder};

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

fn main() {
    server::new(|| {
        App::new()
            .resource("/", |r| r.f(greet))
            .resource("/{name}", |r| r.f(greet))
    })
    .bind("127.0.0.1:8000")
    .expect("Can not bind to port 8000")
    .run();
}
```

Plaintext

Best plaintext responses per second, Dell R440 Xeon Gold + 10 GbE (272 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	actix-raw	7,040,642	100.0%	0	Plt	Rus	Non	act	Lin	Rea
2	fasthttp	7,026,716	99.8%	0	Plt	Go	Non	Non	Lin	Rea
3	ulib-plaintext_fit	7,016,745	99.7%	0	Plt	C++	Non	ULi	Lin	Rea
4	wizzardo-http	7,014,982	99.6%	0	Mcr	Jav	Non	Non	Lin	Rea
5	libreactor	7,011,500	99.6%	0	Mcr	C	Non	Non	Lin	Rea
6	ulib	7,008,767	99.5%	0	Plt	C++	Non	ULi	Lin	Rea
7	tokio-minihttp	7,007,335	99.5%	0	Mcr	Rus	Rus	tok	Lin	Rea
8	may-minihttp	7,006,770	99.5%	0	Mcr	Rus	Rus	may	Lin	Rea
9	rapidoid	6,999,211	99.4%	0	Plt	Jav	Rap	Non	Lin	Rea
10	rapidoid-http-fast	6,995,006	99.4%	0	Plt	Jav	Rap	Non	Lin	Rea
11	aspcore	6,970,937	99.0%	0	Plt	C#	.NE	kes	Lin	Rea
12	actix	6,715,989	95.4%	0	Mcr	Rus	Non	act	Lin	Rea
13	hyper	6,254,812	88.8%	0	Mcr	Rus	Rus	Hyp	Lin	Rea
14	cpoll_cppsp	5,734,965	81.5%	0	Plt	C++	Non	Non	Lin	Rea
15	h2o	5,326,290	75.7%	0	Plt	C	Non	Non	Lin	Rea
16	proteus	4,841,612	68.8%	0	Mcr	Jav	Utw	Non	Lin	Rea
17	netty	4,560,356	64.8%	0	Plt	Jav	Nty	Non	Lin	Rea
18	immutant	4,178,241	59.3%	0	Mcr	Clj	Utw	Non	Lin	Rea
19	revenj	3,911,182	55.6%	0	Ful	C#	Non	Non	Lin	Rea
20	act	3,858,510	54.8%	0	Ful	Jav	Utw	Non	Lin	Rea

Actix: routing & extraction

```
App::with_state(api.clone())
    .route("/", Method::GET, fetch)
```

```
pub fn fetch(
    log: RequestLogger,
    state: State<ApiContext>,
    req: Query<FetchRequest>)
    -> impl Responder {

    slog_info!(log, "Fetching comments");

    risso_api::fetch(&state, req.into_inner())
        .map(Json)
        .responder()
}
```

```
https://risso.rs/?uri=/blog/great-post
```

```
#[derive(Debug, Deserialize)]
pub struct FetchRequest {
    uri: String,
    parent: Option<CommentId>,
    limit: Option<i32>,
    nested_limit: Option<usize>,
    after: Option<DateTime<Utc>>,
    plain: Option<i32>,
}
```

The power of Rust generic traits

zero cost abstraction

```rust
pub trait FromRequest<S> {
    /// Future that resolves to a Self
    type Result: Into<AsyncResult<Self>>;

    /// Convert request to a Self
    fn from_request(req: &HttpRequest<S>)
        -> Self::Result;
}
```

```rust
impl<T> Deref for Query<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.0
    }
}
```

```rust
pub struct Query<T>(T);

impl<T> Query<T> {
    pub fn into_inner(self) -> T {
        self.0
    }
}

impl<T, S> FromRequest<S> for Query<T> {
    type Result = Result<Self, Error>;

    #[inline]
    fn from_request(req: &HttpRequest<S>) -> Self::Result {
        serde_urlencoded::from_str::<T>(req.query_string())
            .map_err(|e| e.into())
            .map(Query)
    }
}
```

Path extraction with tuples

```
App::with_state(api.clone())
    .route("/id/{id}/unsubscribe/{email}/{key}", Method::GET, unsubscribe)
```

```
fn unsubscribe(state: State<ApiContext>, path: Path<(String, String, String)>)
   -> impl Responder {

   let (id, email, key) = path.into_inner();

   risso_api::unsubscribe(&state, id, email, key)
       .map(Json)
       .responder()
}
```

DIESEL

Guides API Docs

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.

Get Started View on Github

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.

Diesel: struct - relational mapping

Similar to JOOQ in Java:

- define your schema
- define associated data structures
- write strongly-typed SQL in a Rust DSL

Also handles migrations

Works with the r2d2 connection pool

Diesel: schema

```
table! {
    comments (id) {
        #[sql_name = "tid"]
        thread_id -> Integer,
        id -> Integer,
        parent -> Nullable<Integer>,
        created -> Double,
        modified -> Nullable<Double>,
        mode -> Integer,
        remote_addr -> Text,
        text -> Text,
        author -> Nullable<Text>,
        email -> Nullable<Text>,
        website -> Nullable<Text>,
        likes -> Integer,
        dislikes -> Integer,
        notification -> Bool,
    }
}
```

```
table! {
    threads (id) {
        id -> Integer,
        uri -> Text, // Unique
        title -> Text,
    }
}

joinable!(comments -> threads (thread_id));
```

```
#[derive(Queryable)]
pub struct Thread {
    pub id: i32,
    pub uri: String,
    pub title: String,
}
```

Diesel: query

```
SELECT comments.parent,count(*)
   FROM comments INNER JOIN threads ON
       threads.uri=? AND comments.tid=threads.id AND
       (? | comments.mode = ?) AND
       comments.created > ?
   GROUP BY comments.parent
```

```
// Comment count for main thread and all reply threads for one url.
let stmt = comments::table
    .inner_join(threads::table)
    .select((comments::parent, count_star()))
    .filter(threads::uri.eq(uri)
       .and(CommentMode::mask(mode))
       .and(comments::created.gt(after)),
    ).group_by(comments::parent);

trace!("{:?}", diesel::debug_query(&stmt));

let result = stmt.load(cnx);
```

Diesel: use a thread pool!

```
let future_comments = ctx.spawn_db(move |cnx| {
   comments::table.inner_join(threads::table)
        .select(comments::all_columns)
        .load(cnx)
});

future_comments.map(|comments| {
   // render markdown, etc
});
```

Diesel: use a thread pool!

```
let thread_pool = tokio_threadpool::Builder::new()
   .name_prefix("risso-api")
   .keep_alive(Some(std::time::Duration::from_secs(30)))
   .pool_size(config.max_connections as usize)
   .build();
```

```
pub fn spawn_db<F, T, E>(&self, f: F)
    -> impl Future<Item = T, Error = failure::Error>
    where
        E: std::error::Error,
        F: FnOnce(&Connection) -> Result<T, E>,
{
   oneshot::spawn_fn(
       move || {
           let cnx = cnx_pool.get()?;
           f(&cnx)
       },
       &self.self.thread_pool.sender(),
   )
}
```

Logs

logs: de facto standard

- simple API, lots of appenders/backends (log4rs 😱)
- exposes a set of macros

```
let env = env_logger::Env::default()
   .filter_or(env_logger::DEFAULT_FILTER_ENV, "info");

env_logger::Builder::from_env(env).init();
error!("Oops");
warn!("Attention");
info!("Starting...");
debug!("Been there");
trace!("{:?}", diesel::debug_query(&q));
```

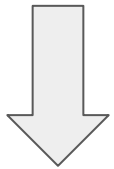
```
ERROR 2018-11-07T17:52:07Z: risso_api::models: Oops
 WARN 2018-11-07T17:52:07Z: risso_api::models: Attention
 INFO 2018-11-07T17:52:07Z: risso_api::models: Starting...
```

slog: structured logs

```
let json_drain = slog_json::Json::default(std::io::stderr());
let drain = drain.filter_level(Level::Info);

let drain = slog_async::Async::new(drain.fuse()).build().fuse();

let log = Logger::root(
   drain,
   slog_o!("location" => FnValue(|info : &Record| {
            format!("{}:{}", info.module(), info.line())
   })),
);
```

slog: structured logs


```
{"msg":"Starting...","level":"INFO","ts":"2018-11-07T18:58:12.077454+01:00",
      "location":"risso_actix:110"}
{"msg":"Using database at temp/comments.db with max 10 connections.",
       "level":"INFO","ts":"2018-11-07T18:58:12.083808+01:00",
       "location":"risso_api::context:36"}
{"msg":"Starting 8 workers","level":"INFO","ts":"2018-11-07T18:58:12.094084+01:00","
       location":"actix_net::server::server:201"}
{"msg":"Starting server on 127.0.0.1:8080","level":"INFO","ts":"2018-11-07T18:58:12.106399+01:00",
       "location":"actix_net::server::server:213"}
```

slog: tracing requests

```
App::with_state(api.clone())
    .route("/", Method::GET, fetch)
    .middleware(actix_web_requestid::RequestIDHeader)
```

```
curl -v localhost:8080/?uri=/blog/great-post/

HTTP/1.1 200 OK
content-type: application/json
request-id: fSBClUEnHy
```

slog: tracing requests

```rust
impl<S> FromRequest<S> for RequestLogger {
    fn from_request(req: &HttpRequest<S>) -> Self::Result {
        let new_log = slog_scope::logger().new(o!("request_id" => req.request_id()));
        Ok(RequestLogger(new_log))
    }
}
```

```rust
pub fn fetch(log: RequestLogger, ...) -> impl Responder {
    slog_info!(log, "Fetching comments");
    ...
```

```
{"msg":"Fetching comments","level":"INFO","ts":"2018-11-08T09:27:54.266083+01:00",
      "request_id":"fSBClUEnHy","location":"risso_actix:91"}
```

Monitoring: prometheus

```
impl MetricsMiddleware {
    pub fn new() -> Result<MetricsMiddleware, failure::Error> {

        let histogram_opts = HistogramOpts::new("req_time", "http processing time");
        let histogram = HistogramVec::new(histogram_opts, &["status"])?;

        registry::register(Box::new(histogram.clone()))?;
        Ok(MetricsMiddleware { histogram })
    }
}
```

```
        let secs = duration_to_seconds(start.elapsed());

        self.histogram
            .with_label_values(&[response.status().as_str()])
            .observe(secs);
```

Monitoring: prometheus

```
curl -v localhost:8080/metrics

# HELP req_time http processing time
# TYPE req_time histogram
req_time_bucket{status="200",le="0.01"} 9
req_time_bucket{status="200",le="0.1"} 9
req_time_bucket{status="200",le="1"} 9
req_time_bucket{status="200",le="10"} 9
req_time_sum{status="200"} 0.022794493
req_time_count{status="200"} 9
req_time_bucket{status="404",le="0.01"} 1
req_time_bucket{status="404",le="0.1"} 1
req_time_bucket{status="404",le="1"} 1
req_time_bucket{status="404",le="10"} 1
req_time_sum{status="404"} 0.000518249
req_time_count{status="404"} 1
```

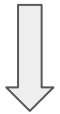
Dockerization

Makefile

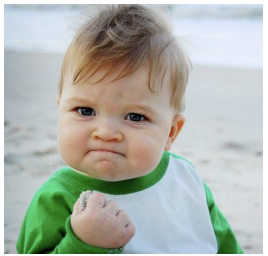
```
muslrust-builder := docker run --rm -it\
 -v $(PWD):/volume\
 -v muslrust-cache:/root/.cargo\
 clux/muslrust:1.30.0-stable

build-musl:
  $(muslrust-builder) cargo build \
     --package risso_actix --release
  $(muslrust-builder) strip --only-keep-debug \
     target/release/risso_actix

docker-image: build-musl
  docker build -t risso-actix .
```

Dockerfile

```
FROM scratch

WORKDIR /risso
COPY target/release/risso_actix .
CMD ["/risso/risso_actix"]
```



8 MB!!!



Conclusion

- Are we web yet? Yes!
- If it compiles, it runs (minus logic bugs)
- It's fast
- It's small

- Rust is different: you have to learn it
- async / await is coming, frameworks will have to adapt

What's next?

- Finish the code…

- Serverless
  rust-aws-lambda: port of the AWS Go SDK
  Web Assembly running in node

- Administration/moderation front-end
  yew - a React-like Rust framework to build SPAs in Web Assembly

Thanks!

Questions? Let’s meet outside!

elastic - stickers!