Start Building microservices with Rust

Pierre-Henri Symoneaux

Code::Dive 2017

Wroclaw – 2017-10-15

Introduction

Is Rust a good choice for microservices ?

What are microservices

Using the basic building blocks in Rust

NOKIA

NOKIA

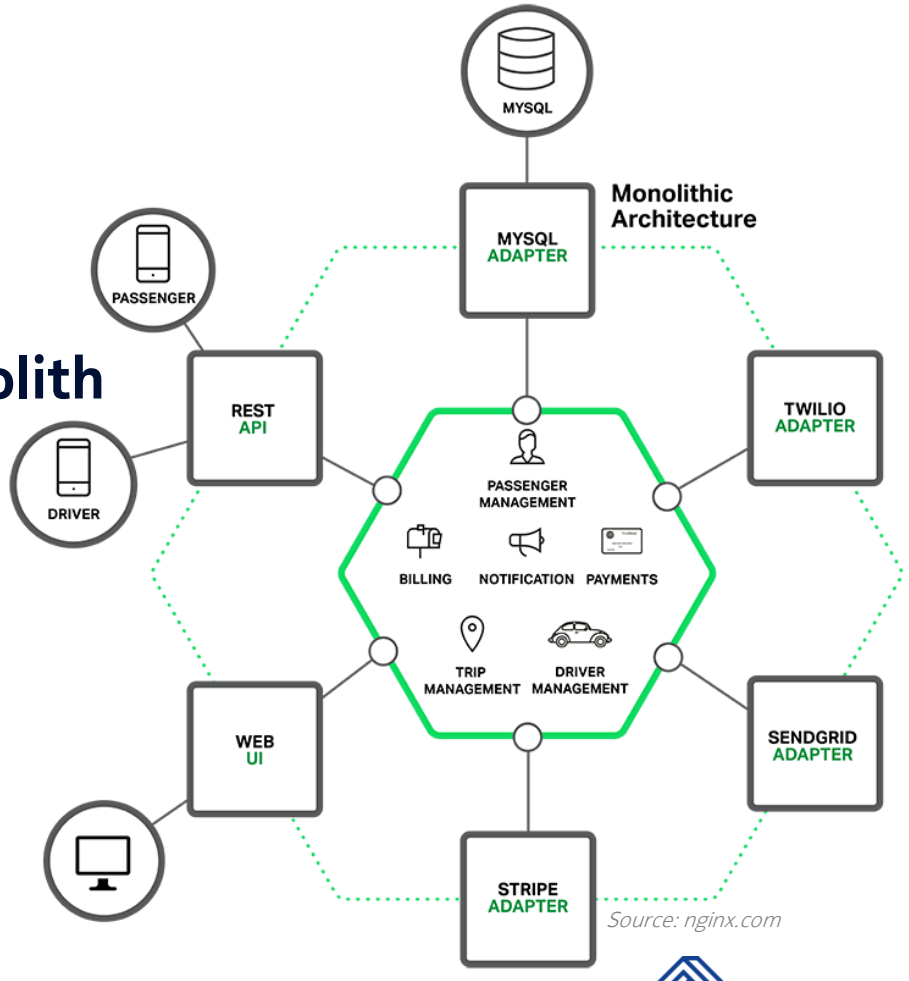
Microservices architecture

A quick refresher

Microservices architecture
Introduction

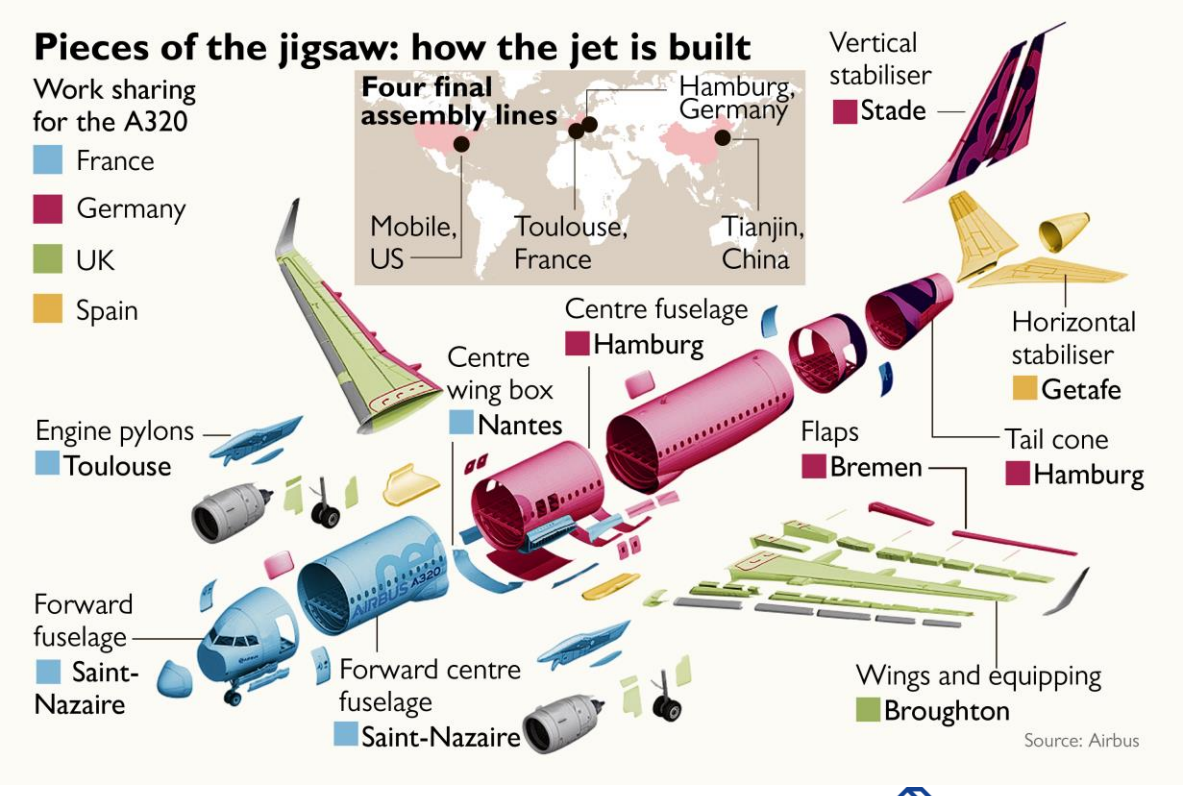
Microservices architecture
The classical monolith architecture

- More time = **more code**
- Few years = **monstruous monolith**
- Development = **pain**
- Maintenance = **pain**
- Refactoring = **pain**
- Build time = **pain**
- On-boarding = **pain**
- → **Big Ball Of Mud**



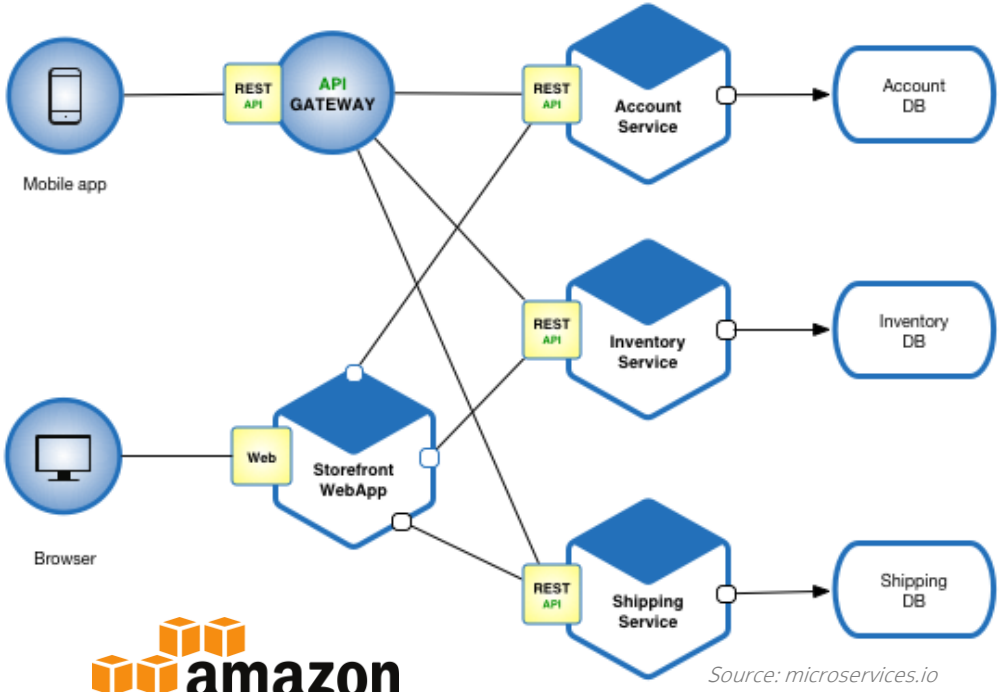
Microservices architecture
A different way of working

- Split the work in smaller independant pieces
- The simpler is better
- Integration by DevOps



Microservices architecture
Let's split the monolith

- Separate processes
- Smaller and simpler
- With distinct functionalities
- Distributed over the network
- Decoupled from each other
- Stateless & share-nothing
- Scalables
- Fault tolerant



Source: microservices.io

➔ Fits in cloud dynamic nature

Adopted by **Google**, **Netflix**, **Amazon**, **Uber** and many more

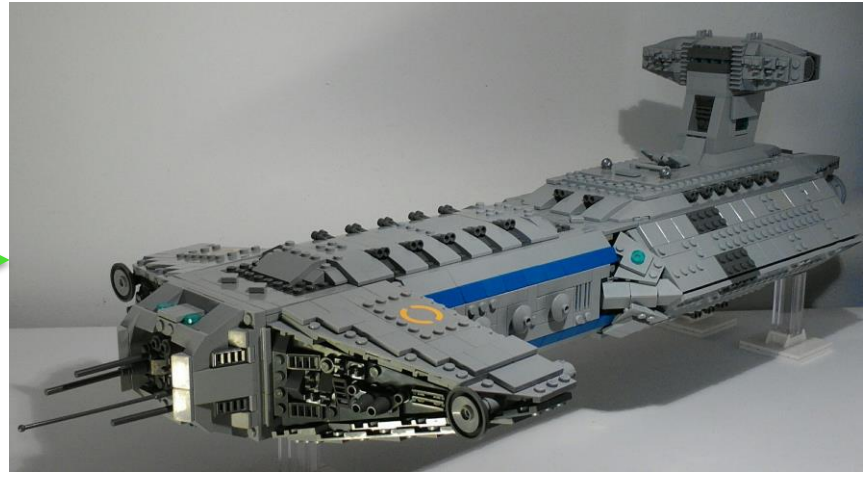
Microservices architecture
Not a silver bullet

- Several drawbacks and issues

- Discipline is required

Microservices architecture
Don't mess with your interfaces

To get from **that** to **that**
You need …



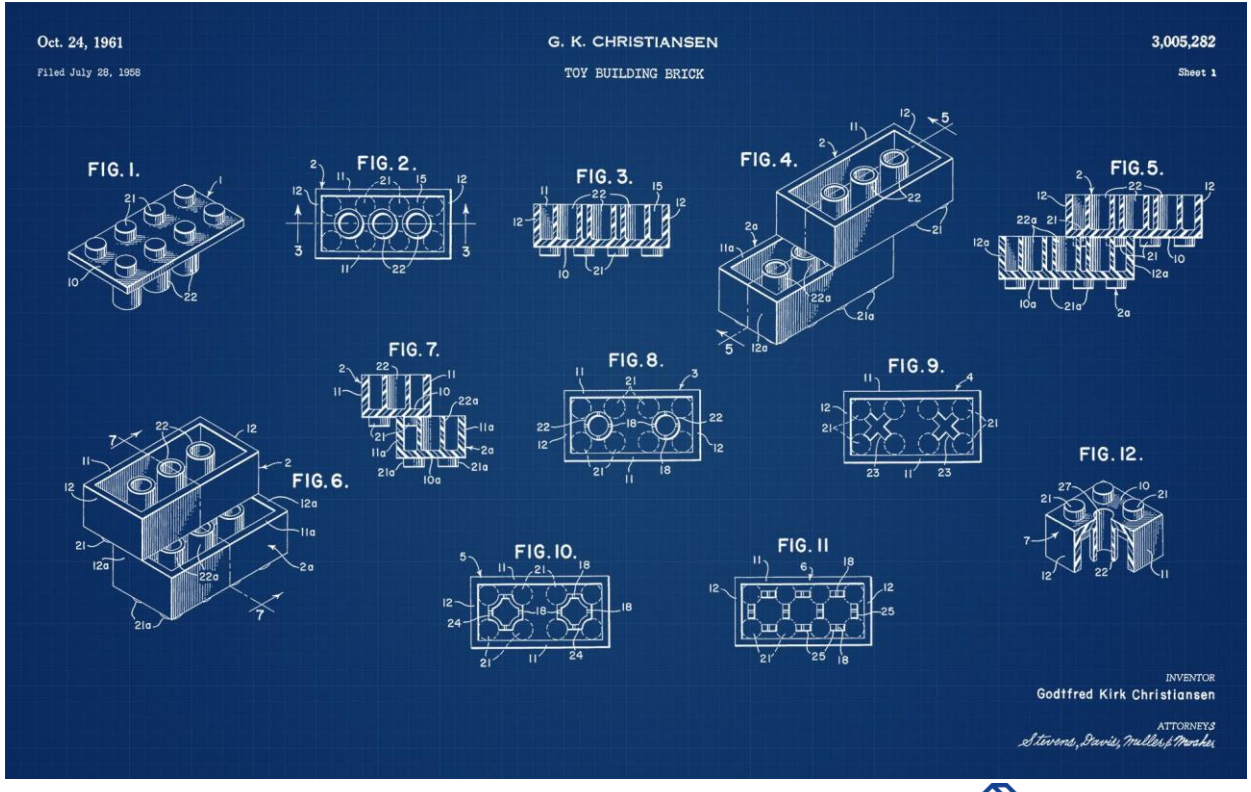
Microservices architecture

Don’t mess with your interfaces

Contracts

Interfaces

Best practices

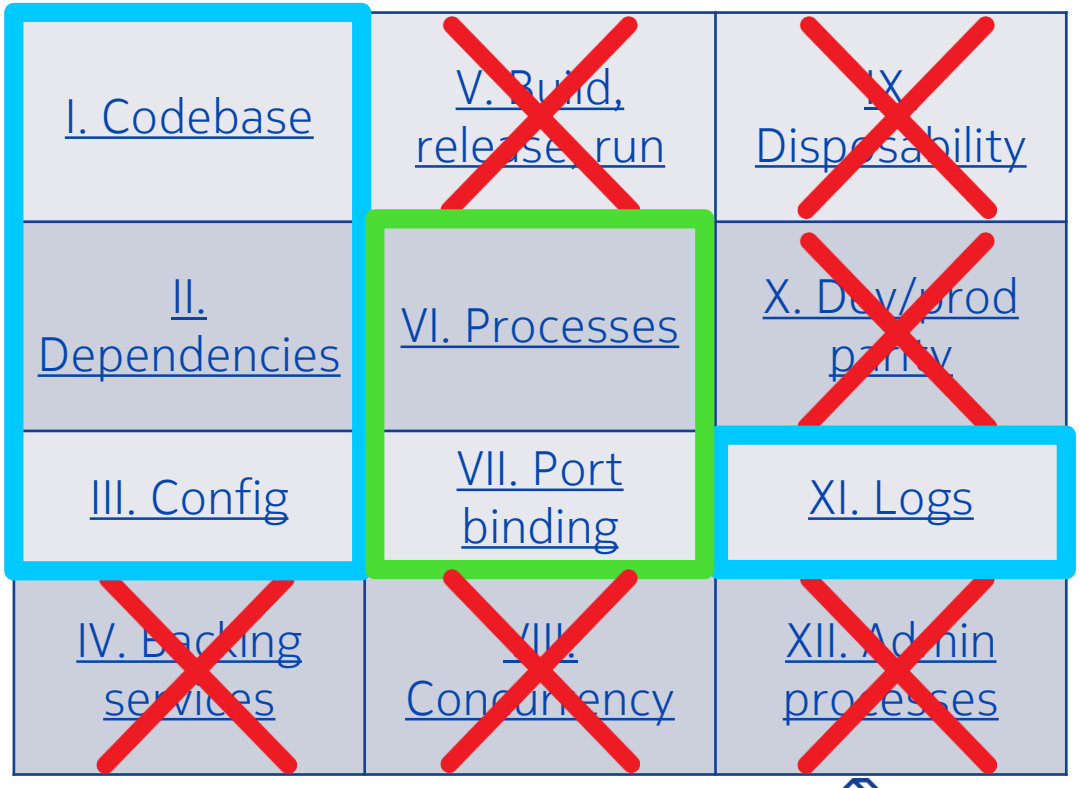


The Twelve-Factor app

12 rules to successfully design Software as a Service

- Use **declarative** formats for setup automation
- **Clean contract** with the OS
- **Maximum portability**
- Suitable for **deployment** on **cloud platforms**
- **Minimize** dev/prod **divergence**
- Enabling **continuous deployment**
- Easy **scaling**

- https://12factor.net/



NOKIA

Let’s dive into the code

Applying the rules while using Rust

The language in a few words

- Strongly typed
- Compiled
- Bare-metal (no bytecode)
- No Garbage Collector
- A powerful and very strict compiler performing many checks for safety

Major features are :

- Zero-cost abstractions
- Move semantic
- Guaranteed **memory safety**
- Threads without data races
- Trait-based generics
- Pattern matching
- Type inference
- Minimal runtime
- Efficient C bindings

NOKIA

Microservices in Rust
Why not Rust

- Hard to learn
- Young and immature ecosystem
- The asynchronous style is still maturing

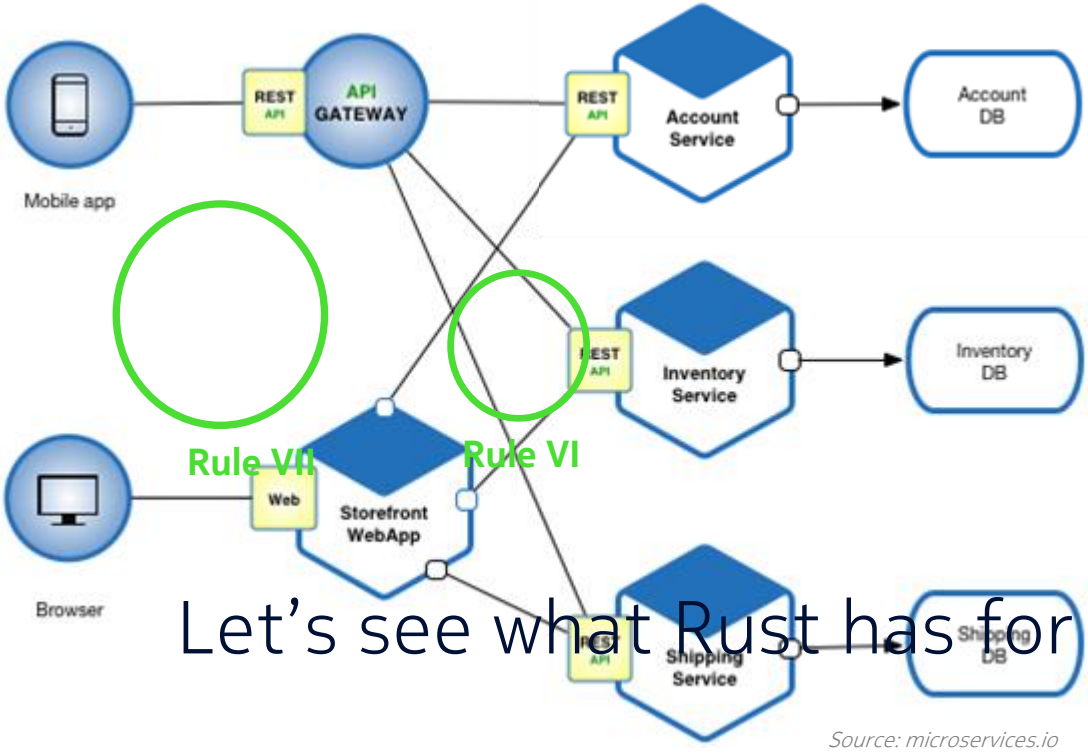
NOKIA

Microservices in Rust
Why choosing Rust

- Performance and low memory footprint
- Thread and memory safety
- Easy packaging and distribution
- Expressive and productive language
- Portable code

NOKIA

Let’s study a simple case



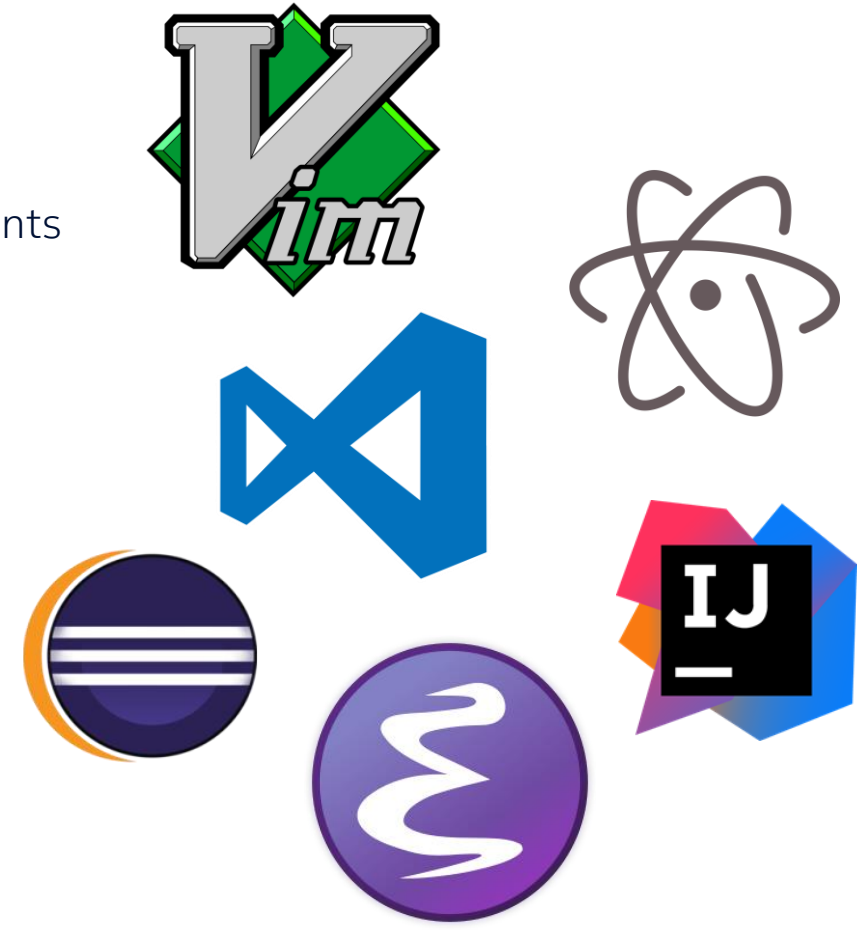
Let’s see what Rust has for us

Source: microservices.io

NOKIA

Rust and microservices
Setting up a development environment

- **Rustup** to install and manage rust environments
- **Rustc** the Rust compiler
- **Cargo** the package management tool
- **Rustfmt** to format the code
- **Clippy** the rust code linter
- **Racer** or **RLS** for code completion
- **Kcov** for code coverage
- **Lldb** for debugging
- And a good **code editor** with a Rust plugin
  - VS Code, Atom, IntelliJ, Eclipse, Vim, Emacs



NOKIA

NOKIA

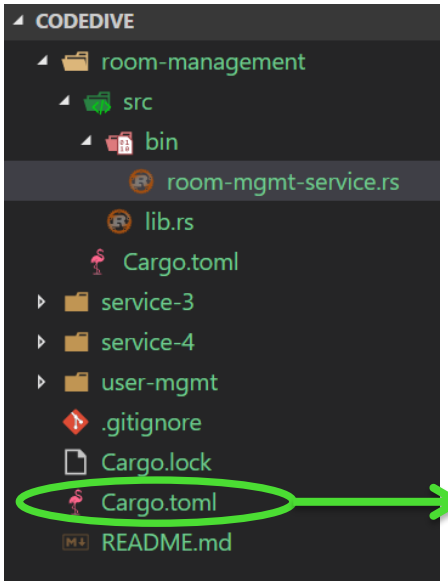
Let’s start with the basis

I	II	III	XI	VI	VII

Rule I : The codebase

One codebase tracked in revision control

- Keep consistency between codebase and deployments
- **Cargo** with a **workspace** is the way to go
  - One subdir per microservice
  - Common libraries in dedicated subdirs
  - Workspace members declared in **Cargo.toml**
- Test all with `cargo test`
- Build all with `cargo build [--release]`




```
λ ls -1 target/debug/*.exe
target/debug/room-mgmt-service.exe
target/debug/service3.exe
target/debug/service4.exe
target/debug/usermgmt-service.exe
```

```
# Content of Cargo.toml
[workspace]
members = [
      "room-management",
      "user-mgmt",
      "service-3",
      "service-4"
]
```


I II III XI VI VII

NOKIA

Rule II : The dependencies
Explicitly declare and isolate dependencies

- Again, **Cargo** is the way to go
- Each service declares its dependencies in its own **Cargo.toml**
- Use of Semantic Versioning
- A **Cargo.lock** is generated
  - **Must to be versioned** with code

user-mgmt/Cargo.toml

```
[package]
name = "user-mgmt"
version = "0.1.0"
# Additional metadata like author, description, etc…

[dependencies]
log = "^0.3"
dotenv = "^0.10"
# Diesel ORM
diesel = {version = "^0.16", features = ["postgres"]}
diesel_codegen = {version = "^0.16", features = ["postres"]}
# Iron web framework
iron = "^0.6"
router = "^0.6"
# Serialization / Deserialization
serde = "^1.0"
serde_json = "^1.0"
serde_derive = "^1.0"
```


NOKIA

Rule III : The configuration
Store config in the environment

Read an environment variable's value with Standard libray

```
use std::env;
// (...)
env::var("DATABASE_URL").expect("DATABASE_URL not set");
```

```
match env::var(key) {
     Ok(val) => println!("{}: {:?}", key, val),
     Err(e) => println!("Error with var {}: {}", key, e)
}
```

```
env.var(key).unwrap_or("default value");
```

- Boost your development & testing with the **dotenv** crate
  - Loads environment variables from a **.env** file, if available

```
extern crate dotenv;
// (...)
dotenv::dotenv().unwrap();
// Env variables are now
updated with content from
.env file, if available
```


NOKIA

Rule XI : Logs
Treat logs as event streams

```
INFO:usermgmt_service: User management service starting
INFO:user_mgmt::dao: Connecting to database: user-mgmt/mydb.db
DEBUG:hyper::server: threads = 32
DEBUG:hyper::server: Incoming stream
DEBUG:hyper::server: Incoming stream
DEBUG:hyper::server::request: Request Line: Get AbsolutePath("/users") Http11
```

- Logs → STDOUT or STDERR
- Continuous flow
- Delegate capture, aggregation, storage to execution environment
- Crates (libraries) :
  - **Log**: Logging facade
    - Used by apps & libraries
  - **Env_logger**:
    - Log to **stderr**
    - Configured with env variables *(**Rule III**)*
    - EG:
      - **`export RUST_LOG=debug`**
      - **`export RUST_LOG=warn,myservice=debug`**

```rust
#[macro_use] extern crate log;
(...)
error!("This is an error")
warn!("Warning");
info!("User management service starting");
debug!("What about debugging");
trace!("Never ever activate this in prod");
```

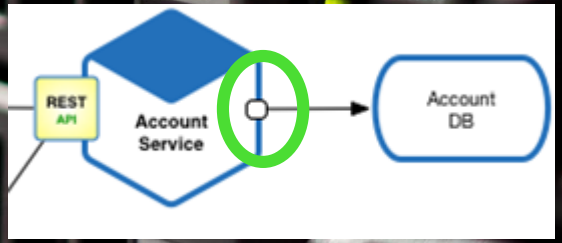
```rust
extern crate env_logger;
// (...)
dotenv::dotenv().unwrap(); // Optional
env_logger::init().unwrap();
```

NOKIA

Rule VI : Processes

Execute the app as one or more stateless processes

I | II | III | XI | VI | VII



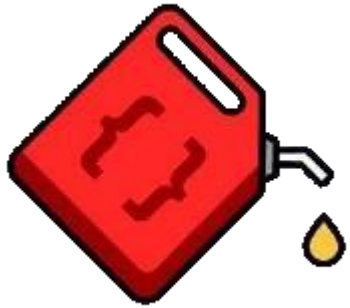
Using external storage
What's available

- SQL support:
  - **Postgres**: Safe wrapper ![downloads 80k](https://img.shields.io/badge/downloads-80k-brightgreen.svg)
  - **Rusqlite**: Safe wrapper ![downloads 58k](https://img.shields.io/badge/downloads-58k-brightgreen.svg)
  - **Mysql**: Pure rust ![downloads 31k](https://img.shields.io/badge/downloads-31k-brightgreen.svg)

- NoSQL
  - **Redis**: pure Rust ![downloads 39k](https://img.shields.io/badge/downloads-39k-brightgreen.svg)
  - **Mongodb**: pure Rust ![downloads 9k](https://img.shields.io/badge/downloads-9k-brightgreen.svg)
  - **Cdrs**: Cassandra, pure Rust ![downloads 5k](https://img.shields.io/badge/downloads-5k-brightgreen.svg)

- **Memcached** has some libraries too

Diesel, the best Rust ORM out there
Set it up



- Supports **sqlite**, **postgres**, and **mysql,** with **transactions**
- Support for **migrations**
- Code generated from infered schema at compile time
- Good level of documentation

Add the dependencies

```
diesel = {version = "^0.16", features = ["postgres"]}
diesel_codegen = {version = "^0.16", features = ["postgres"]}
```

Install diesel CLI tool

Setup your database

```
# cargo install diesel_cli --no-default-features --features "postgres"
# echo DATABASE_URL=postgres://username:password@localhost/diesel_demo > .env
# diesel setup
# diesel migration generate create_users
# diesel migration run
```

And connect to it

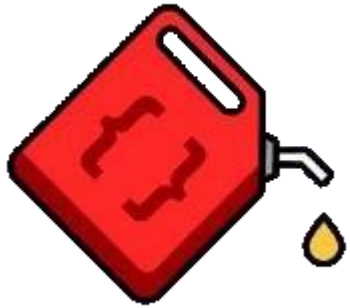
```
let db_url = env::var("DATABASE_URL").expect("DATABASE_URL not set");
info!("Connecting to database: {}", db_url);
let connection = PgConnection::establish(&db_url).expect("Cannot connect to DB");
```


I II III XI VI VII

NOKIA

Diesel, the best Rust ORM out there

Map your data



schema.rs

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

SQL defined in migration files

```
CREATE TABLE users (
      id INTEGER NOT NULL PRIMARY KEY
      AUTOINCREMENT,
      username VARCHAR NOT NULL,
      firstname VARCHAR NOT NULL,
      lastname VARCHAR NOT NULL
);
```

```
table! {
      users (id) {
            id -> Integer,
            username -> Text,
            firstname -> Text,
            lastname -> Text,
      }
}
```

Auto-implement **Queryable** trait

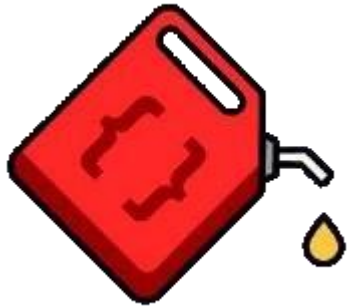
Row to **struct** conversion

```
use schema::users;
#[derive(Queryable, Debug)]
pub struct User {
      pub id: i32,
      pub username: String,
      pub firstname: String,
      pub lastname: String
}
```

```
#[derive(Insertable, Debug)]
#[table_name="users"]
pub struct NewUser<'a> {
      pub username: &'a str,
      pub firstname: &'a str,
      pub lastname: &'a str
}
```

Struct to **row** conversion

Diesel, the best Rust ORM out there
Querying & pooling



- Basic querying

```
use schema::users::dsl::*;
use models::*;

users.load::<User>(&connection).unwrap();

let user = NewUser{/*...*/};
diesel::insert(user)
       .into(users)
       .execute(&connection)
       .unwrap();

users.filter(username.eq("codedive"))
     .first(&connection)
     .optional()
     .unwrap();

diesel::delete(users.filter(id.eq(12)))
       .execute(&connection)
       .unwrap();
```

- Pool your connections with R2D2

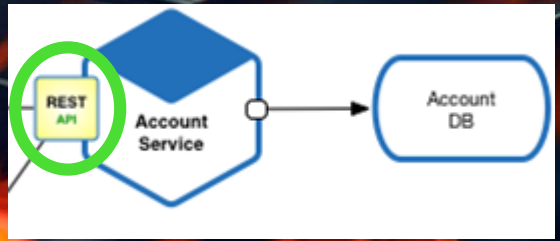
```
use r2d2;
use r2d2_diesel::ConnectionManager;
/* (...)
let cfg = r2d2::Config::default();
let mgr = ConnectionManager::new(db_url);

let pool = r2d2::Pool::<ConnectionManager<PgConnection>>
                      ::new(cfg, mgr)
                       .expect("Failed to create pool");

let conn = pool.get().unwrap();
users.load::<User>(&*conn).unwrap();
// conn will get back to pool when reaching end of scope
```


NOKIA

Rule VII : Port binding

Export services via port binding

I | II | III | XI | VI | VII



Data exchange
Serialization and deserialization support

- **Serde** 

  - No runtime reflection
  - Format agnostic

- Format supported
  - **JSON**: serde_json 

  - **YAML**: serde_yaml 

  - **MessagePack**: rmp 

  - **TOML**: toml 

  - **BSON**: bson 

  - **XML**: serde_xml 

  - More ...

```
extern crate serde;
#[macro_use] extern crate serde_derive;
```

```
#[derive(Queryable, Serializable, Deserializable)]
pub struct User {
      pub id: int32,
      /*..*/
}
```

Example with JSON

```
extern crate serde_json;
/*(...)*/
serde_json::to_string_pretty(&user).unwrap();
```

Build a web interface
What is available

- Low level HTTP
  - **hyper.rs** : 

    - HTTP client and server library
    - Used by almost all frameworks
    - Previously sync, now async (since v0.11)
- Higher level framework
  - **Iron** : 

    - Composable framework
    - Synchronous
  - **Rocket** : 

    - Very powerful
    - Very well documented
    - Uses macros and compiler plugins → Nightly Rust only

NOKIA

Build a web interface
Iron framework : The basis

- Easy to bootstrap
  - **Cargo.toml** dependency `iron = "^0.6"`
  - **main.rs**

```
extern crate iron;
use iron::prelude::*;
use iron::status;
fn main() {
     Iron::new(|_: &mut Request| {
          Ok(Response::with((status::Ok, "Hello World!")))
     }).http("localhost:3000").unwrap();
}
```

  - Then browse to http://localhost:3000
- Easy to extend
  - Let's add some routing

I II III XI VI VII

NOKIA

Build a web interface
Iron framework : Add some routing

- Create some handlers

```
fn list_users(req: &mut Request) -> IronResult<Response> {
     Ok(Response::with((status::Ok, "List users")))
}

fn find_user(req: &mut Request) ->IronResult<Response> {
     let id = req.extensions.get::<Router>().unwrap()
                            .find("id").unwrap();
     let reply = format!("Find user with id = {}", id);
     Ok(Response::with(s))
}
```

- Add some routes to them

```
extern crate router;
let mut routes = router::Router::new();
routes.get("/users", list_users, "list_users");
routes.get("/user/:id", find_user, "find_user");
Iron::new(routes).http("localhost:3000").unwrap();
```

Dependency

```
router = "^0.6"
```

I II III XI VI VII

NOKIA

Build a web interface
Iron framework : Add some state

- Remember the pool of Diesel connections ?

```
pub type ConnectionPool = r2d2::Pool<ConnectionManager<PgConnection>>;
let pool: ConnectionPool = /*(...)*/
```

- Let's add some middlewares

```
let mut chain = Chain::new(routes);
chain.link_before(move |r: &mut Request| {
      r.extensions.insert::<ConnectionPool>(pool.clone());
      Ok(())
});
Iron::new(chain).http("localhost:3000").unwrap();
```

- And use it from handlers

```
fn list_users(req: &mut Request) -> IronResult<Response> {
      let pool = req.extensions.get::<ConnectionPool>().unwrap();
      let conn = pool.get().unwrap();
      let usrs = users.load::<User>(&*conn).unwrap();
      let s = serde_json::to_string(&usrs).unwrap();
      Ok(Response::with((status::Ok, s)))
}
```

I II III XI VI VII

NOKIA

Build a web interface
What about Rocket

- Less boilerplate

```
let pool: ConnectionPool = /*(...)*/;
Rocket::ignite()
        .manage(pool)
        .mount("/", routes![list_users, find_user]);
```

Dependencies injection
Parameters validation and injection

```
#[get("/user/<user_id>")]
fn find_user(pool: State<ConnectionPool>, user_id: i32) -> Json<User> {
     /*fetch the user from database & send it back in json*/
}
```

Nightly rust
Unstable features

I II III XI VI VII

NOKIA

What about HTTP clients

Reqwest, a high level HTTP client

- Ease of use
- Fetching data

```
let users = reqwest.get("http://my.service.net/users")
                   .json::<Vec<User>>()?;
```

```
reqwest.get("http://my.service.net/user/2")
       .json::<User>()?;
```

- Deleting data

```
let client = reqwest::Client::new();
let res = client.delete("http://my.service.net/user/2")
               .send()?;
```

- Posting data

```
let new_user = NewUser {
      username: "rustacean",
      firstname: "Rust",
      lastname: "Language"
};
```

```
client.post("http://my.service.net/user")
      .json(&new_user)
      .send()?;
```


I II III XI VI VII

NOKIA

Conclusion

- We just scratched the surface
- There is much more about microservices
- Rust has everything needed to start
- Asynchronous style is still maturing
- And with stabilization, growth of community, it will become a serious challenger against other languages

NOKIA

NOKIA


NOKIA

NOKIA

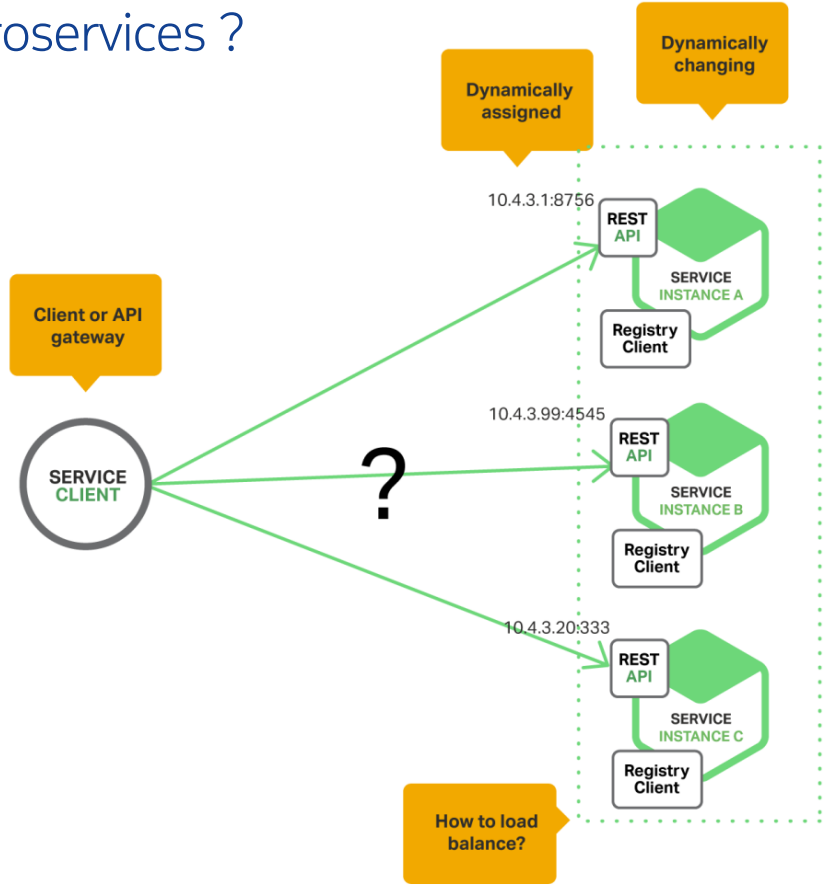
Backup

Concurrency

Scale out with the model process

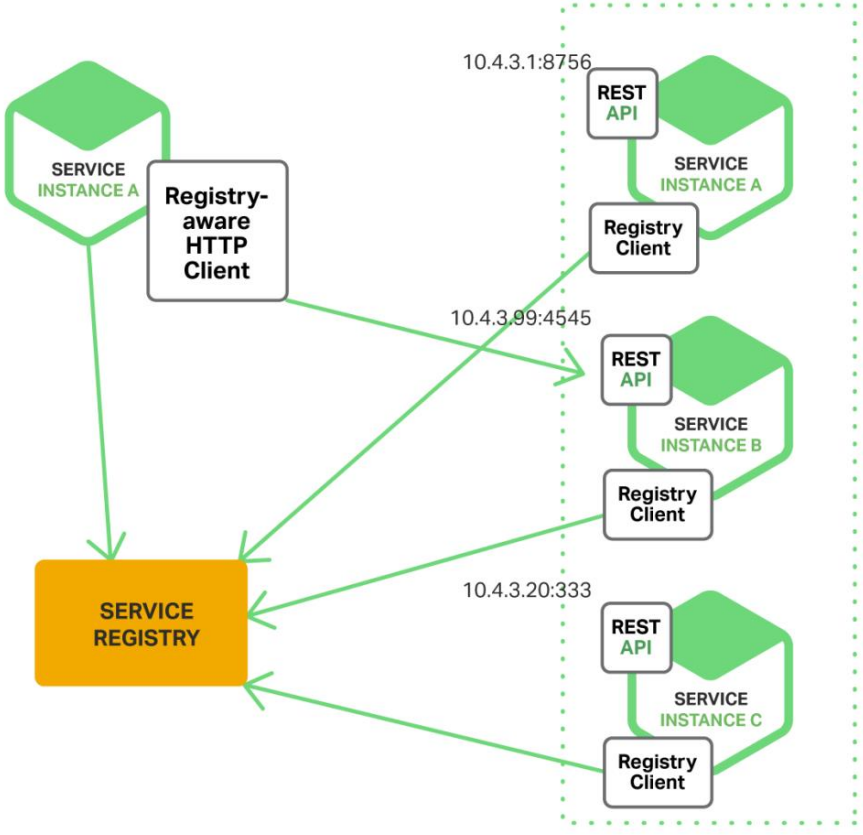
How to manage connections between microservices ?
The problem

- Highly dynamic environment
- Number of instances may change very fast
- IP addresses and ports do too



How to manage connections between microservices ?
Solution 2 : Using a service discovery

- **Consul** : Synchronous, lack documentation
- **Zookeeper**: Synchronous, well documented
- **Etcd** : Async, based on futures and tokio. Well documented. V2 API supported, v3 is planned.

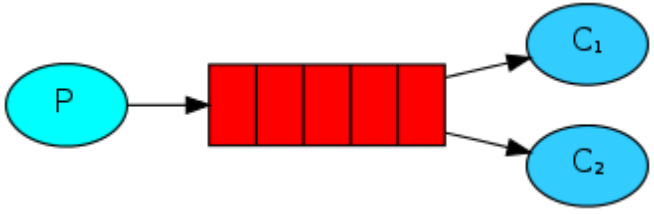


How to manage connections between microservices ?
Solution 3 : Use a messaging middleware

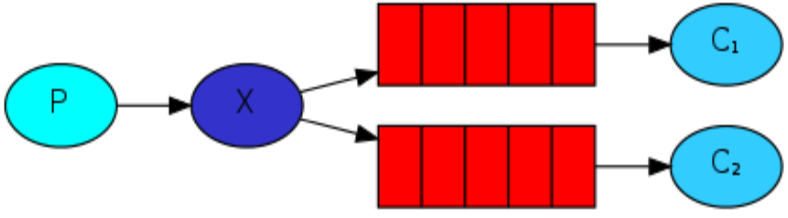
- Microservices are connected to the message broker servers
- They don’t need to know each other
- They produce and read messages to/from named channels

→ Highly decoupled application

Procuder/consumers



Publish/subscribe

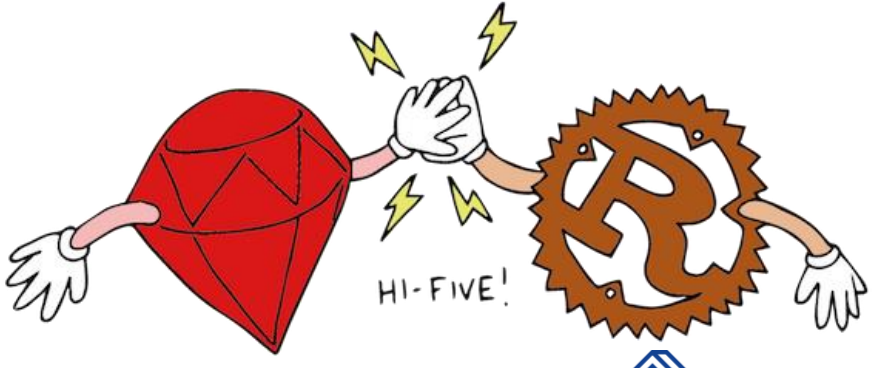


Message broker support
Kafka, RabbitMQ, mqtt

- RabbitMQ
  - **Amqp**
  - **Lapin-futures**
  - **Lapin-async**
- Kafka
  - **Kafka**
- MQTT
  - **mqtt-protocol**

Mixing Rust with other languages

- Possibility to bind with other languages for
  - Performances
  - Workaround in missing lib
  - Embedding (glue with a script)
  - C
  - Ruby / mRuby
  - Python
  - Java
  - Lua



NOKIA

What about not doing some REST
A quick overview

- Blocking TCP/UDP with Standard library
- **Mio** : Non-blocking TCP/UDP 

- **Tokio** : Event loop and async IO 

- **Protobuf** : Google protocol buffer 

- **Capnp** : Capnproto 

- **Juniper :** GraphQL **juniper** 

NOKIA