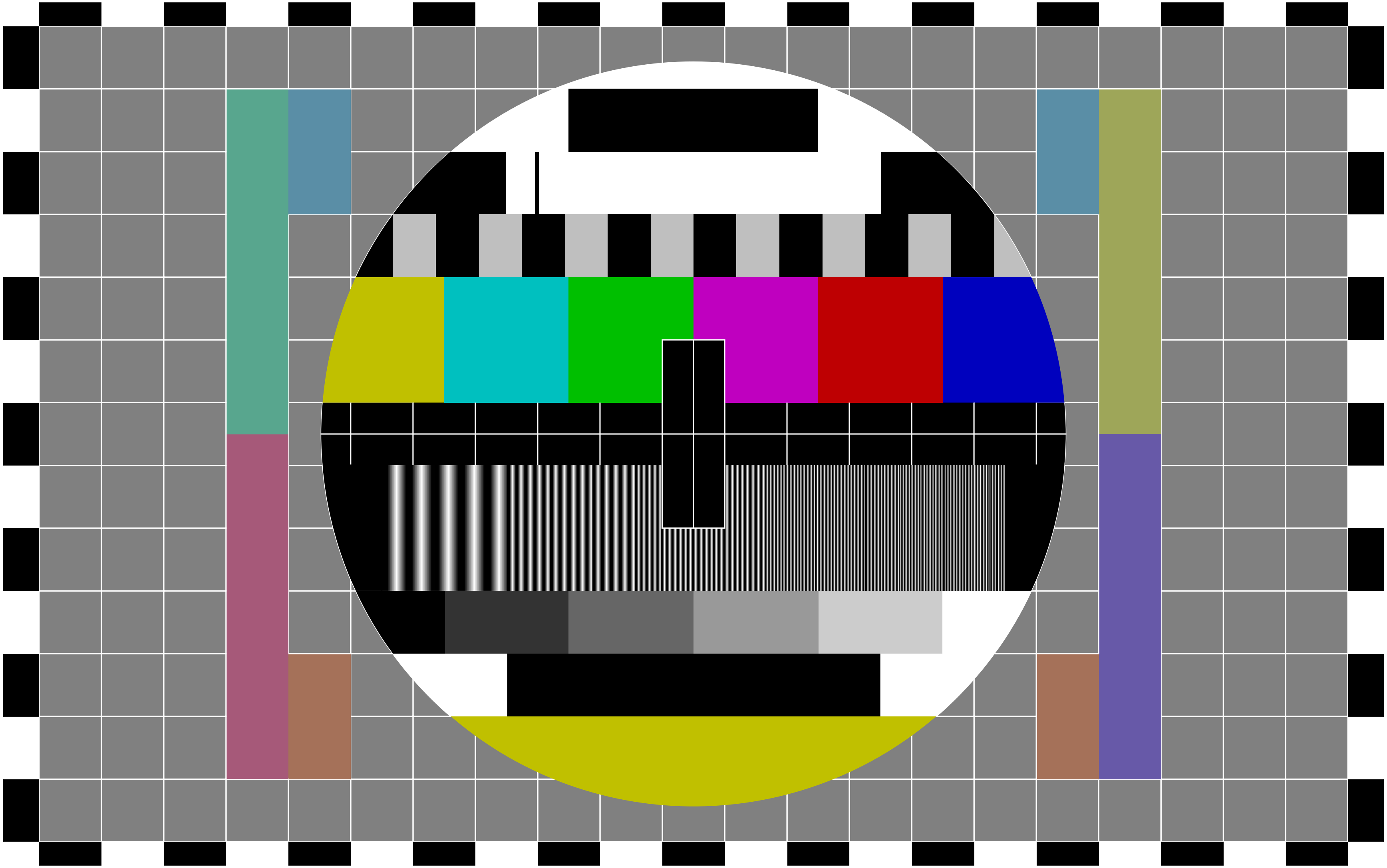
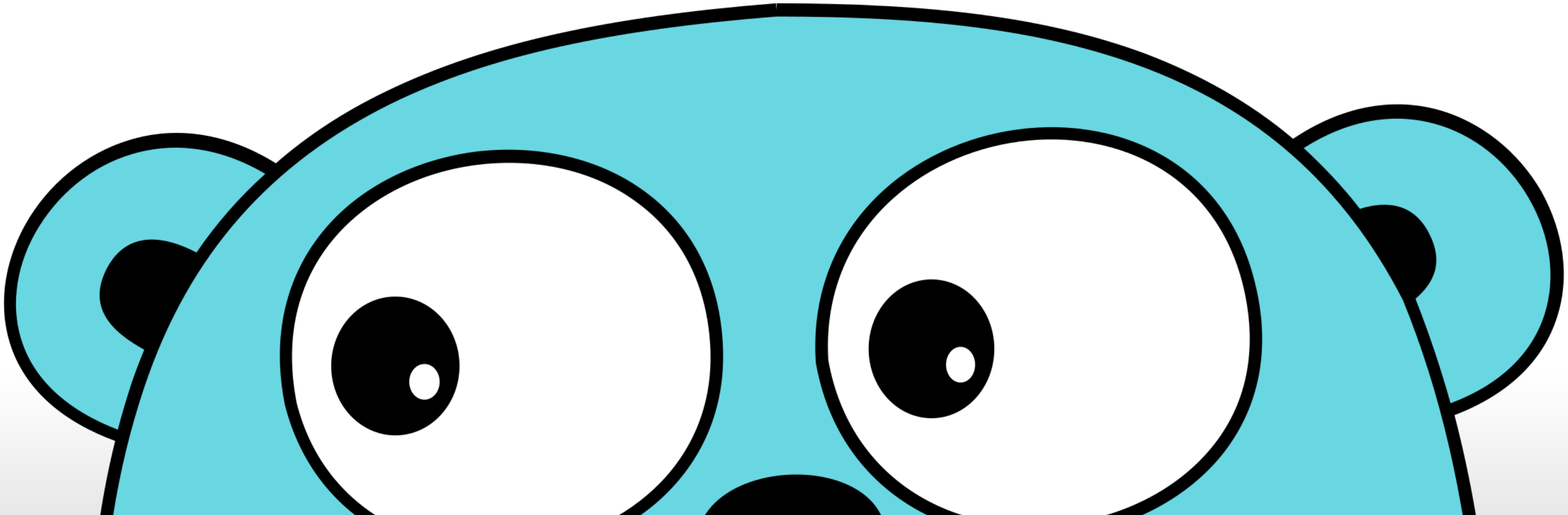


Why I switched from Rust to Go

... and why I'm never going back



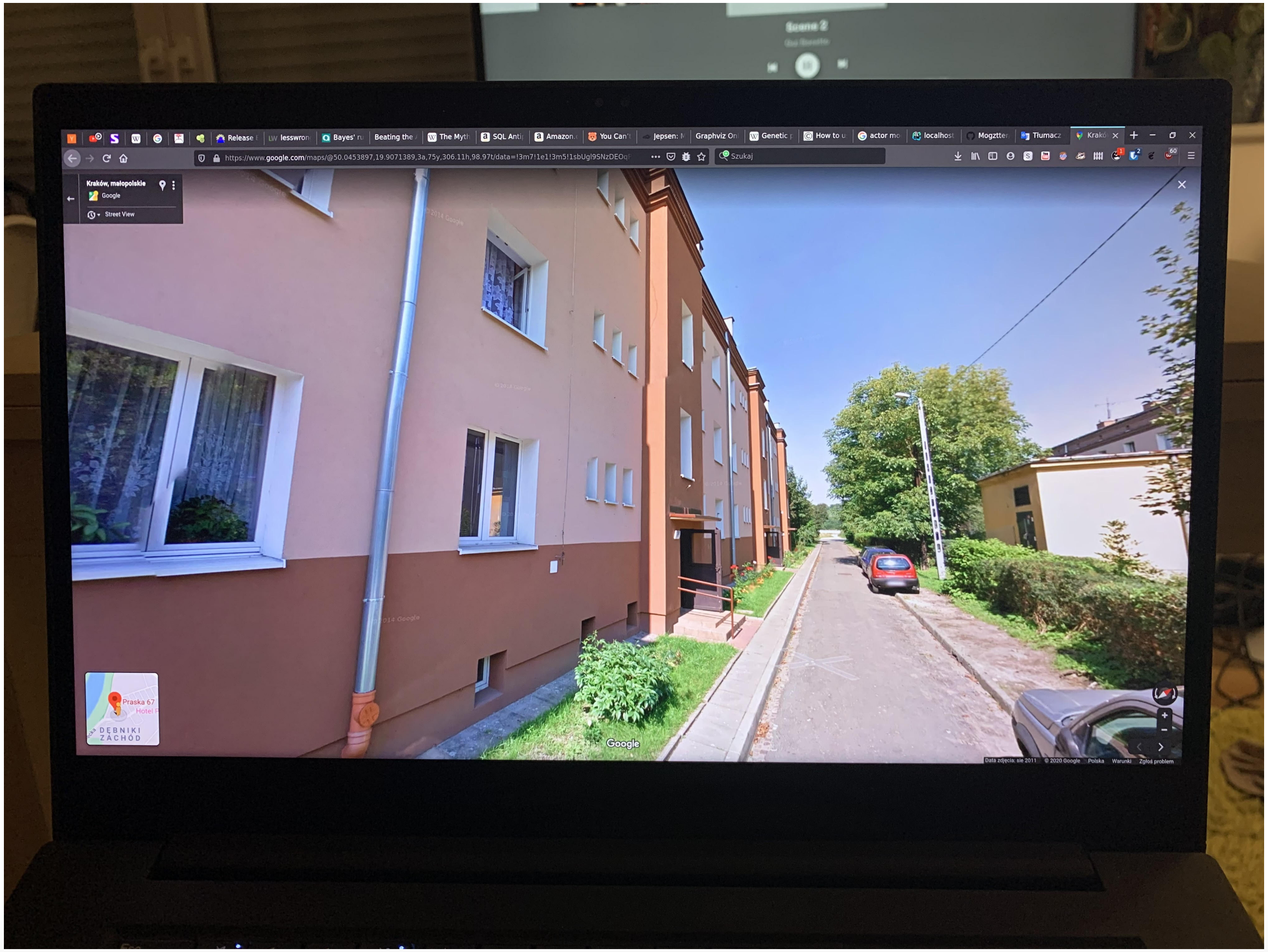
Fantastic Actors and Where to Find Them

building a simple async actor system from scratch

Patryk Wychowaniec

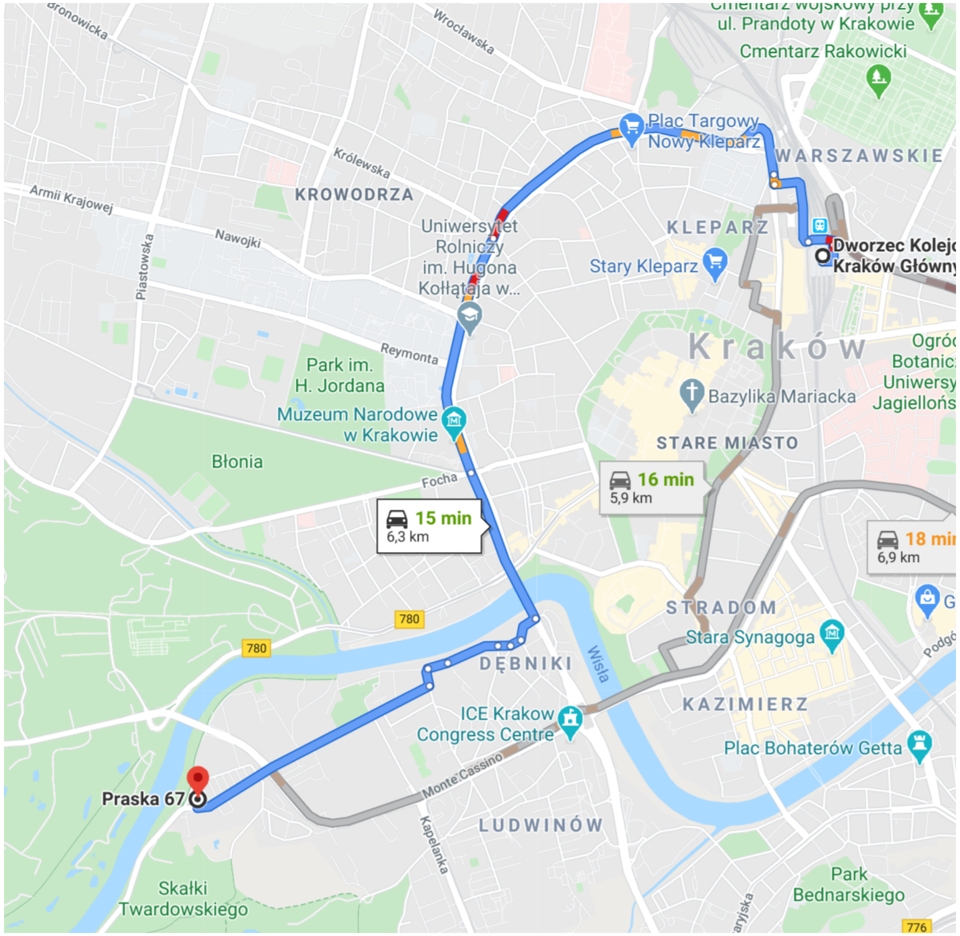
ENODRIVER

I moved out of Kraków to Wrocław about 6 months ago



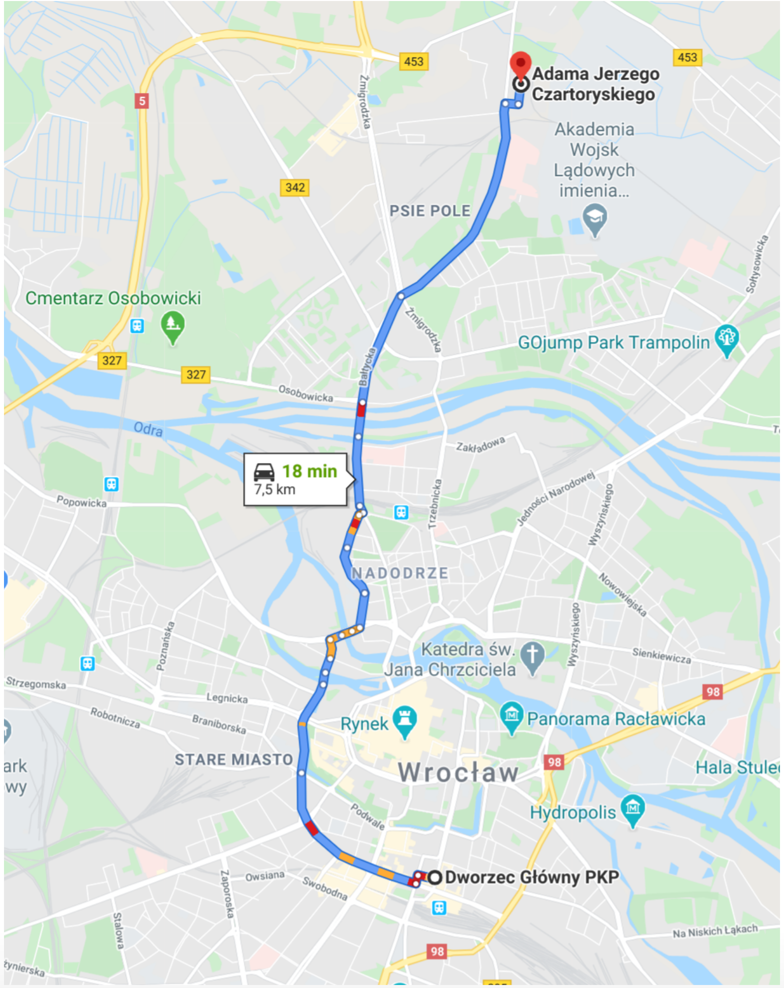
ENODRIVER

While in Kraków, I’ve been living on semi-outskirts of the city



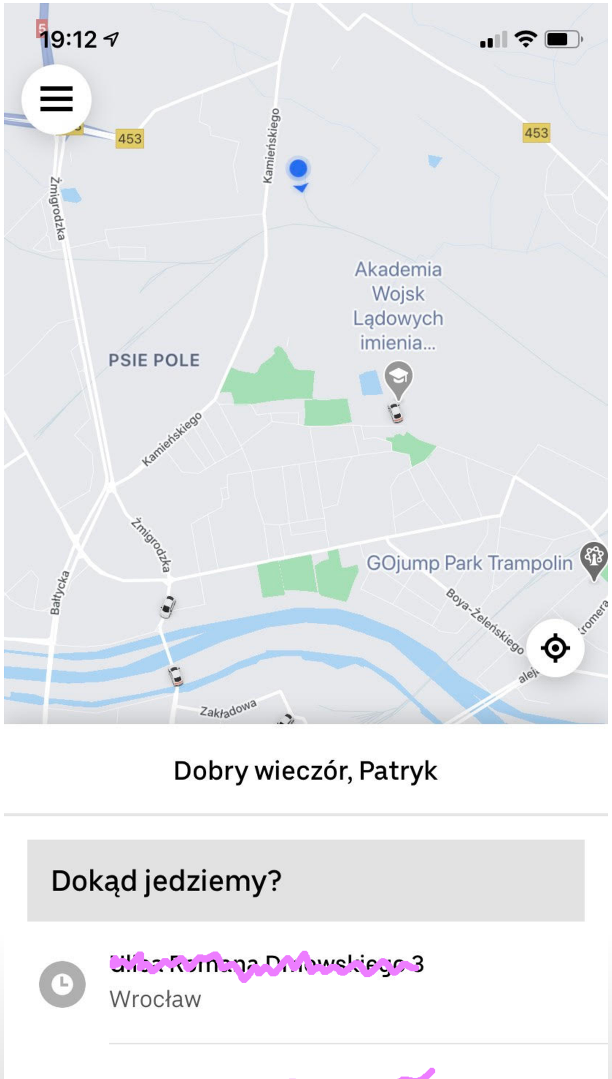
ENODRIVER

… and right now, I consider my location similar



ENODRIVER

What's totally different though, is the taxi service



Dobry wieczór, Patryk

Dokąd jedziemy?

[redacted]
Wrocław

ENODRIVER

Solution: Let’s roll our own!

What will we be creating?

Me, Myself and I

My name's **Patryk Wychowaniec**, a.k.a. **Patryk27**:

keybase.io/patryk27

reddit.com/u/patryk27

github.com/patryk27

4programmers.net (patryk27)

Crash course

What’s an actor?

NOTE

Actor is a self-contained **object** that can receive **messages** and **act** upon them.

Crash course

What’s an actor?

Actors are usually used to model what’s commonly known as the *service layer*, e.g.:

RUST

```
fn main() {
    let mut mw = Microwave::new();

    mw.put("popcorn-kernels");
    mw.put("iphone");
    mw.start(Duration::from_secs(60));
}
```

Crash course

What's an actor?

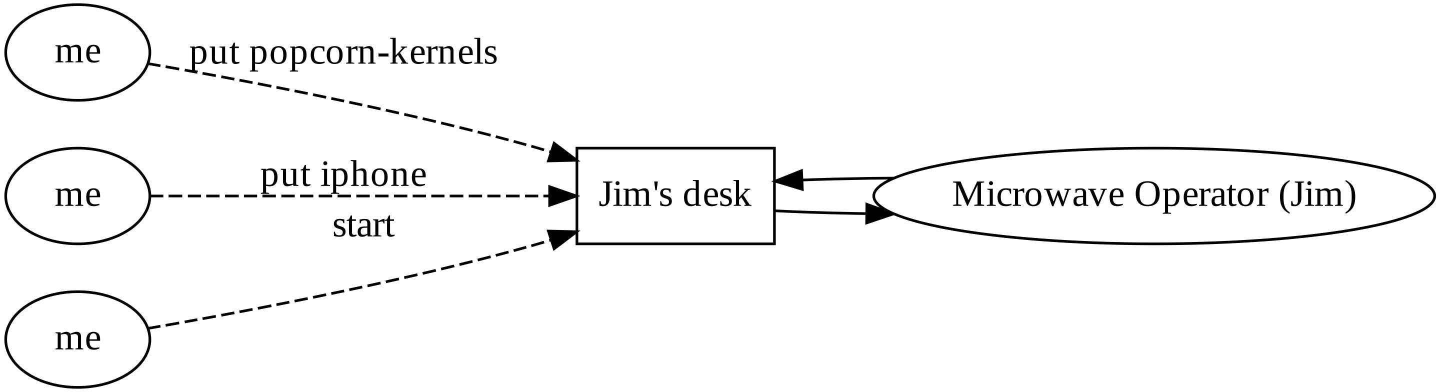
RUST

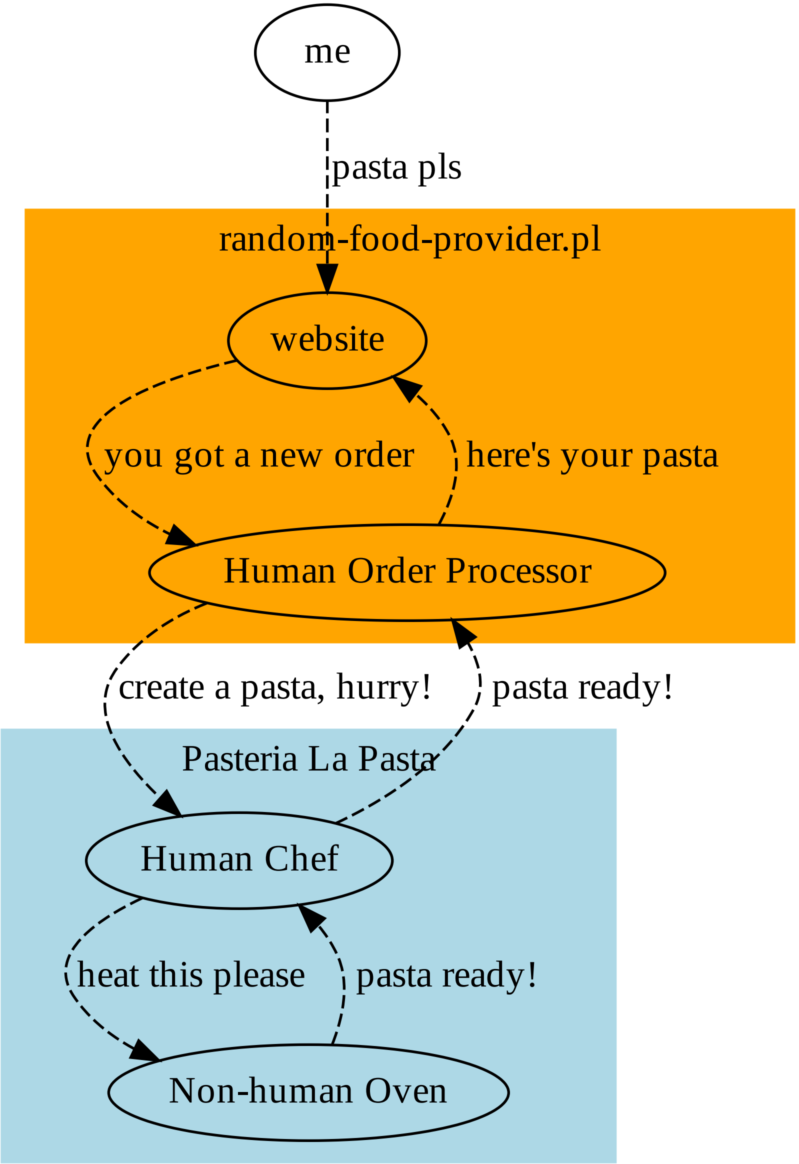
```rust
fn main() {
    let mut mw = MicrowaveOperator::new();

    mw.slip_note("please put these `popcorn-kernels` inside");
    mw.slip_note("please put this `iphone` inside");
    mw.slip_note("please wave micros for 60 seconds");
}
```

Crash course

What’s an actor?




me
pasta pls
random-food-provider.pl
website
you got a new order
here's your pasta
Human Order Processor
create a pasta, hurry!
pasta ready!
Pasteria La Pasta
Human Chef
heat this please
pasta ready!
Non-human Oven

Crash course

How can we model an actor in Rust?

Since we don’t have any power over actor’s control flow, how can we let it know we need its service?

Crash course

How can we model an actor in Rust?

Using **channels**!

RUST

```rust
use tokio::stream::StreamExt;
use tokio::sync::mpsc::unbounded_channel;
use tokio::task;

#[tokio::main]
async fn main() {
    let (tx, mut rx) = unbounded_channel();

    task::spawn(async move {
        while let Some(msg) = rx.next().await {
            println!("recv: {}", msg);
        }

        println!("tx dropped");
    });

    let _ = tx.send("hello");
    let _ = tx.send("darkness");
    let _ = tx.clone().send("my");
    let _ = tx.clone().send("old");
    let _ = tx.send("friend");
}
```

TIP

`tx` stands for `transmitter`, `rx` stands for `receiver`

Crash course

Channels

RUST

```rust
use tokio::stream::StreamExt;
use tokio::sync::mpsc::unbounded_channel;
use tokio::task;

#[tokio::main]
async fn main() {
    let (tx, mut rx) = unbounded_channel();

    task::spawn(async move {
        while let Some(msg) = rx.next().await {
            println!("recv: {}", msg);
        }

        println!("tx dropped");
    });

    let _ = tx.send("hello");
    let _ = tx.send("darkness");
    let _ = tx.clone().send("my");
    let _ = tx.clone().send("old");
    let _ = tx.send("friend");
}
```

```
recv: hello
recv: darkness
recv: my
recv: old
recv: friend
tx dropped
```

Crash course

Channels

Table 1. Channels (as in `tokio 0.2`)

Name	# of producers	# of consumers	# of messages
oneshot	one	one	one
mpsc	many	one	many
broadcast	many	many	many
watch	one	many	many

Baby steps

Simple actor-oriented key-value database

RUST

```rust
#[tokio::main]
async fn main() {
    println!("Hello, World!");
}
```

RUST

```rust
pub enum DatabaseMsg {
    /* ... */
}
```

RUST

```
pub enum DatabaseMsg {
    Put {
        /* ... */
    },

    Get {
        /* ... */
    },

    List {
        /* ... */
    },
}
```

RUST

```rust
pub struct DatabaseActor {
    /* ... */
}

impl DatabaseActor {
    pub fn new() -> Self {
        /* ... */
    }

    pub async fn start(mut self, mut mailbox: mpsc::UnboundedReceiver<DatabaseMsg>) {
        /* ... */
    }
}
```

RUST

```rust
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    pub fn new() -> Self {
        Self { items: Default::default() }
    }

    pub async fn start(mut self, mut mailbox: mpsc::UnboundedReceiver<DatabaseMsg>) {
        /* ... */
    }
}
```

RUST

```rust
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    pub fn new() -> Self {
        Self { items: Default::default() }
    }

    pub async fn start(mut self, mut mailbox: mpsc::UnboundedReceiver<DatabaseMsg>) {
        while let Some(msg) = mailbox.next().await {
            self.handle_msg(msg).await;
        }
    }

    pub async fn handle_msg(&mut self, msg: DatabaseMsg) {
        /* ... */
    }
}
```

RUST

```rust
#[tokio::main]
async fn main() {
    let (tx, rx) = mpsc::unbounded_channel();

    task::spawn(
        DatabaseActor::new()
            .start(rx)
    );

    tx.send(/* ... */);
    tx.send(/* ... */);
    tx.send(/* ... */);
    tx.send(/* ... */);
}
```

RUST

```rust
pub enum DatabaseMsg {
    Put {
        /* ... */
    },

    Get {
        /* ... */
    },

    List {
        /* ... */
    },
}
```

RUST

```rust
pub enum DatabaseMsg {
    Put {
        key: String,
        value: String,
    },

    Get {
        /* ... */
    },

    List {
        /* ... */
    },
}
```

RUST

```rust
#[tokio::main]
async fn main() {
    /* ... */

    tx.send(
        DatabaseMsg::Put {
            key: "hello".to_string(),
            value: "world".to_string(),
        },
    );
}
```

RUST

```rust
#[tokio::main]
async fn main() {
    /* ... */

    tx.send(
        DatabaseMsg::Put {
            key: "hello".into(),
            value: "world".into(),
        },
    );
}
```

TIP

Until specialization lands, `.to_string()` invokes all the `std::fmt` machinery and does *not* get optimized into `String::from()`

RUST

```rust
pub enum DatabaseMsg {
    Put {
        /* ... */
    },

    Get {
        key: String,
    },

    List {
        /* ... */
    },
}
```

RUST

```rust
#[tokio::main]
async fn main() {
    /* ... */

    tx.send(
        DatabaseMsg::Get {
            key: "hello".into(),
        },
    );

    // err, how do we read the "returned" value?
}
```

RUST

```rust
pub enum DatabaseMsg {
    Put {
        /* ... */
    },

    Get {
        key: String,
    },

    List {
        /* ... */
    },
}
```

RUST

```rust
pub enum DatabaseMsg {
    Put {
        /* ... */
    },

    Get {
        key: String,
        tx: oneshot::Sender<Option<String>>,
    },

    List {
        /* ... */
    },
}
```

TIP Hollywood principle: *don’t call us, we’ll call you*

RUST

```rust
#[tokio::main]
async fn main() {
    /* ... */

    let (response_tx, response_rx) = oneshot::channel();

    tx.send(
        DatabaseMsg::Get {
            key: "hello".into(),
            tx: response_tx,
        },
    );

    println!("get(\"hello\") = {:?}", response_rx.await);
}
```

RUST

```rust
pub enum DatabaseMsg {
    Put {
        /* ... */
    },

    Get {
        /* ... */
    },

    List {
        tx: oneshot::Sender<Vec<(String, String)>>,
    },
}
```

RUST

```rust
#[tokio::main]
async fn main() {
    /* ... */

    let (response_tx, response_rx) = oneshot::channel();

    tx.send(
        DatabaseMsg::List {
            tx: response_tx,
        },
    );

    println!("list() = {:?}", response_rx.await);
}
```

RUST

```rust
pub enum DatabaseMsg {
    Put {
        key: String,
        value: String,
    },

    Get {
        key: String,
        tx: oneshot::Sender<Option<String>>,
    },

    List {
        tx: oneshot::Sender<Vec<(String, String)>>,
    },
}
```

RUST

```rust
pub enum DatabaseMsg {
    // "tell"-oriented message
    Put {
        key: String,
        value: String,
    },

    // "ask"-oriented message
    Get {
        key: String,
        tx: oneshot::Sender<Option<String>>,
    },

    // "ask"-oriented message
    List {
        tx: oneshot::Sender<Vec<(String, String)>>,
    },
}
```

RUST

```
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    /* ... */

    pub async fn handle_msg(&mut self, msg: DatabaseMsg) {
        /* ... */
    }
}
```

RUST

```
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    /* ... */

    pub async fn handle_msg(&mut self, msg: DatabaseMsg) {
        use DatabaseMsg::*;

        match msg {
            Put { key, value } => {
                /* ... */
            }

            /* ... */
        }
    }
}
```

RUST

```rust
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    /* ... */

    pub async fn handle_msg(&mut self, msg: DatabaseMsg) {
        use DatabaseMsg::*;

        match msg {
            Put { key, value } => {
                self.items.insert(key, value);
            }

            /* ... */
        }
    }
}
```

RUST

```rust
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    /* ... */

    pub async fn handle_msg(&mut self, msg: DatabaseMsg) {
        use DatabaseMsg::*;

        match msg {
            Put { /* ... */ } => {
                /* ... */
            }

            Get { key, tx } => {
                /* ... */
            }

            /* ... */
        }
    }
}
```

RUST

```rust
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    /* ... */

    pub async fn handle_msg(&mut self, msg: DatabaseMsg) {
        use DatabaseMsg::*;

        match msg {
            Put { /* ... */ } => {
                /* ... */
            }

            Get { key, tx } => {
                let value = self.items
                    .get(&key)
                    .cloned();

                tx.send(value);
            }

            /* ... */
        }
    }
}
```

RUST

```rust
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    /* ... */

    pub async fn handle_msg(&mut self, msg: DatabaseMsg) {
        use DatabaseMsg::*;

        match msg {
            Put { /* ... */ } => {
                /* ... */
            }

            Get { /* ... */ } => {
                /* ... */
            }

            List { tx } => {
                /* ... */
            }

            /* ... */
        }
    }
}
```

RUST

```rust
pub struct DatabaseActor {
    items: HashMap<String, String>,
}

impl DatabaseActor {
    /* ... */

    pub async fn handle_msg(&mut self, msg: DatabaseMsg) {
        use DatabaseMsg::*;

        match msg {
            Put { /* ... */ } => {
                /* ... */
            }

            Get { /* ... */ } => {
                /* ... */
            }

            List { tx } => {
                let items = self.items
                    .iter()
                    .map(|(key, value)| {
                        (key.clone(), value.clone())
                    })
                    .collect();

                let _ = tx.send(items);
            }

            /* ... */
        }
    }
}
```

RUST

```rust
#[tokio::main]
async fn main() {
    let (tx, rx) = mpsc::unbounded_channel();

    task::spawn(
        DatabaseActor::new()
            .start(rx)
    );

    // ---- //

    tx.send(DatabaseMsg::Put {
        key: "pizza hut".into(),
        value: "22 536 36 36".into(),
    });

    tx.send(DatabaseMsg::Put {
        key: "telepizza".into(),
        value: "71 321 39 50".into(),
    });

    // ---- //

    let (req_tx, req_rx) = oneshot::channel();

    tx.send(DatabaseMsg::Get {
        key: "telepizza".into(),
        tx: req_tx,
    });

    println!("get(\"telepizza\") = {:?}", req_rx.await.unwrap());
}
```

RUST

```rust
#[tokio::main]
async fn main() {
    let (tx, rx) = mpsc::unbounded_channel();

    task::spawn(
        DatabaseActor::new()
            .start(rx)
    );

    // ---- //

    let tx2 = tx.clone();

    task::spawn(async move {
        loop {
            let (req_tx, req_rx) = oneshot::channel();

            tx.send(DatabaseMsg::Get {
                key: "telepizza".into(),
                tx: req_tx,
            });

            println!("get(\"telepizza\") = {:?}", req_rx.await.unwrap());

            delay_for(Duration::from_millis(100)).await;
        }
    });

    // ---- //

    tx.send(DatabaseMsg::Put {
        key: "telepizza".into(),
        value: "71 321 39 50".into(),
    });
}
```

Baby steps

Growing up

The problem is... our current version is terribly noisy:

RUST

```
let (req_tx, req_rx) = oneshot::channel();

tx.send(DatabaseMsg::Get {
    key: "telepizza".into(),
    tx: req_tx,
});

println!(
    "get(\"telepizza\") = {:?}",
    req_rx.await.unwrap(),
);
```

Baby steps

Growing up

Ideally, we'd like to use it as such:

RUST

```rust
#[tokio::main]
async fn main() {
    let db = Database::new();

    // ----- //

    db.put("pizza hut", "22 536 36 36");
    db.put("telepizza", "71 321 39 50");

    let db2 = db.clone();

    task::spawn(async move {
        db2.put("ozima", "71 338 85 85");
    });

    // ----- //

    println!("get(\"telepizza\") = {:?}", db.get("telepizza").await);
    println!("list() = {:?}", db.list().await);
}
```

RUST

```rust
pub struct Database {
    /* ... */
}

impl Database {
    pub fn new() -> Self {
        /* ... */
    }

    pub fn put(&self, key: impl Into<String>, value: impl Into<String>) {
        /* ... */
    }

    pub async fn get(&self, key: impl Into<String>) -> Option<String> {
        /* ... */
    }

    pub async fn list(&self) -> Vec<(String, String)> {
        /* ... */
    }
}
```

RUST

```rust
pub struct Database {
    /* ... */
}

impl Database {
    pub fn new() -> Self {
        /* ... */
    }

    /* ... */
}
```

RUST

```rust
pub struct Database {
    /* ... */
}

impl Database {
    pub fn new() -> Self {
        let (tx, rx) = mpsc::unbounded_channel();

        task::spawn(
            DatabaseActor::new()
                .start(rx)
        );

        /* ... */
    }

    /* ... */
}
```

RUST

```rust
pub struct Database {
    tx: mpsc::UnboundedSender<DatabaseMsg>,
}

impl Database {
    pub fn new() -> Self {
        let (tx, rx) = mpsc::unbounded_channel();

        task::spawn(
            DatabaseActor::new()
                .start(rx)
        );

        Self { tx }
    }

    /* ... */
}
```

RUST

```rust
#[derive(Clone)]
pub struct Database {
    tx: mpsc::UnboundedSender<DatabaseMsg>,
}

impl Database {
    pub fn new() -> Self {
        let (tx, rx) = mpsc::unbounded_channel();

        task::spawn(
            DatabaseActor::new()
                .start(rx)
        );

        Self { tx }
    }

    /* ... */
}
```

RUST

```rust
#[derive(Clone)]
pub struct Database {
    tx: mpsc::UnboundedSender<DatabaseMsg>,
}

impl Database {
    pub fn new() -> Self {
        /* ... */
    }

    pub fn put(&self, key: impl Into<String>, value: impl Into<String>) {
        /* ... */
    }

    /* ... */
}
```

RUST

```rust
#[derive(Clone)]
pub struct Database {
    tx: mpsc::UnboundedSender<DatabaseMsg>,
}

impl Database {
    pub fn new() -> Self {
        /* ... */
    }

    pub fn put(&self, key: impl Into<String>, value: impl Into<String>) {
        self.tx.send(DatabaseMsg::Put {
            key: key.into(),
            value: value.into(),
        });
    }

    /* ... */
}
```

RUST

```rust
#[derive(Clone)]
pub struct Database {
    tx: mpsc::UnboundedSender<DatabaseMsg>,
}

impl Database {
    pub fn new() -> Self {
        /* ... */
    }

    pub fn put(&self, key: impl Into<String>, value: impl Into<String>) {
        /* ... */
    }

    pub async fn get(&self, key: impl Into<String>) -> Option<String> {
        let (tx, rx) = oneshot::channel();

        self.tx.send(DatabaseMsg::Get {
            key: key.into(),
            tx,
        });

        rx.await.unwrap()
    }

    /* ... */
}
```

RUST

```rust
#[derive(Clone)]
pub struct Database {
    tx: mpsc::UnboundedSender<DatabaseMsg>,
}

impl Database {
    pub fn new() -> Self {
        /* ... */
    }

    pub fn put(&self, key: impl Into<String>, value: impl Into<String>) {
        /* ... */
    }

    pub async fn get(&self, key: impl Into<String>) -> Option<String> {
        /* ... */
    }

    pub async fn list(&self) -> Vec<(String, String)> {
        let (tx, rx) = oneshot::channel();

        self.tx.send(DatabaseMsg::List {
            tx,
        });

        rx.await.unwrap()
    }

    /* ... */
}
```

RUST

```rust
#[derive(Clone)]
pub struct Database {
    tx: mpsc::UnboundedSender<DatabaseMsg>,
}

impl Database {
    pub fn new() -> Self {
        /* ... */
    }

    pub fn put(&self, key: impl Into<String>, value: impl Into<String>) {
        self.tx.send(DatabaseMsg::Put {
            key: key.into(),
            value: value.into(),
        });
    }

    pub async fn get(&self, key: impl Into<String>) -> Option<String> {
        let (tx, rx) = oneshot::channel();

        self.tx.send(DatabaseMsg::Get {
            key: key.into(),
            tx,
        });

        rx.await.unwrap()
    }

    pub async fn list(&self) -> Vec<(String, String)> {
        let (tx, rx) = oneshot::channel();

        self.tx.send(DatabaseMsg::List {
            tx,
        });

        rx.await.unwrap()
    }
}
```

Baby steps

We made it!

RUST

```
#[tokio::main]
async fn main() {
    let db = Database::new();

    // ----- //

    db.put("pizza hut", "22 536 36 36");
    db.put("telepizza", "71 321 39 50");

    let db2 = db.clone();

    task::spawn(async move {
        db2.put("ozima", "71 338 85 85");
    });

    // ----- //

    println!("get(\"telepizza\") = {:?}", db.get("telepizza").await);
    println!("list() = {:?}", db.list().await);
}
```

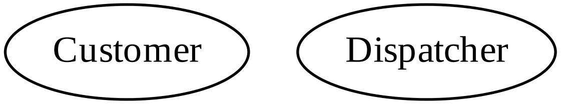
... back to the cabs though...

rusty-cabs

Flow

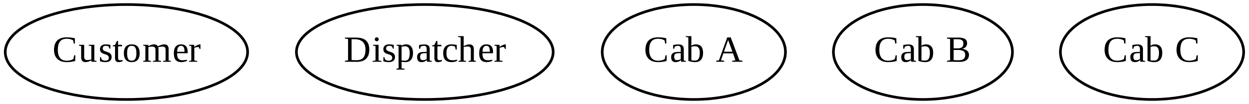
rusty-cabs

Flow



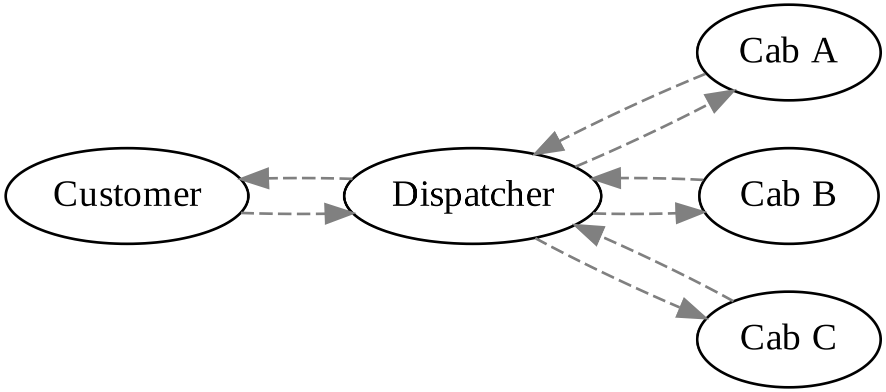
rusty-cabs

Flow



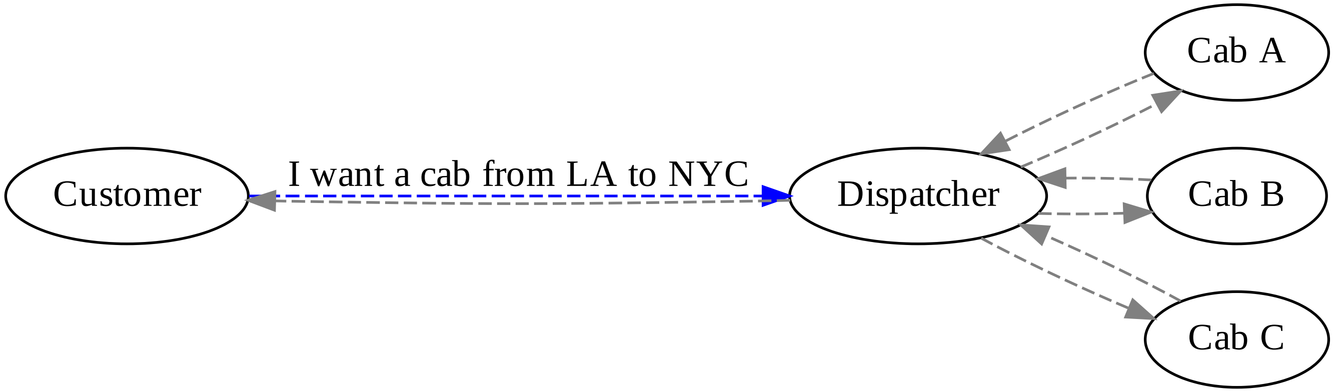
rusty-cabs

Flow



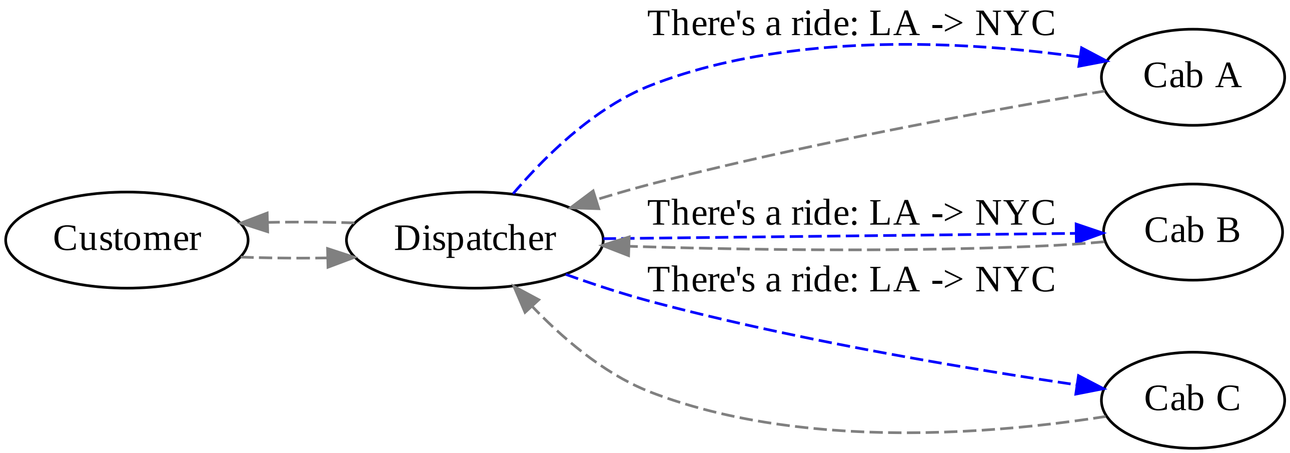
rusty-cabs

Flow



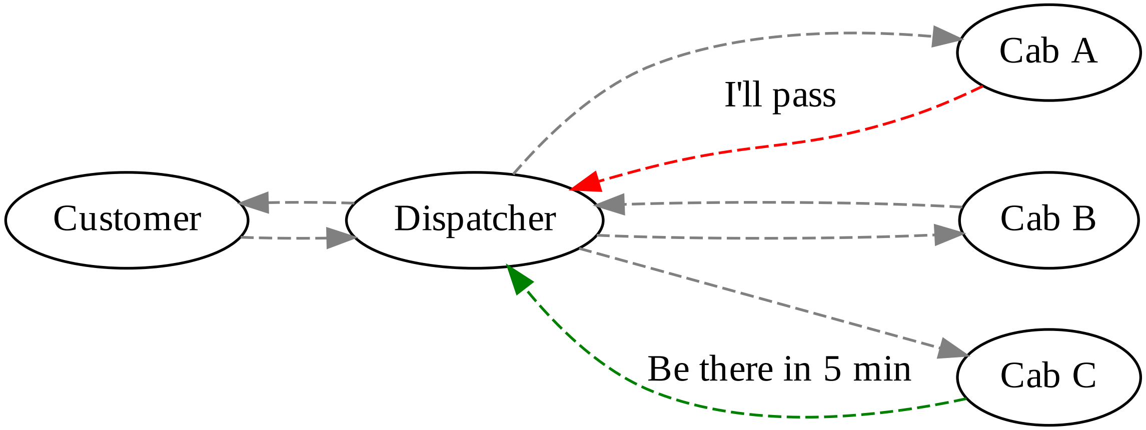
rusty-cabs

Flow


There's a ride: LA -> NYC
Cab A
Customer
Dispatcher
There's a ride: LA -> NYC
Cab B
There's a ride: LA -> NYC
Cab C

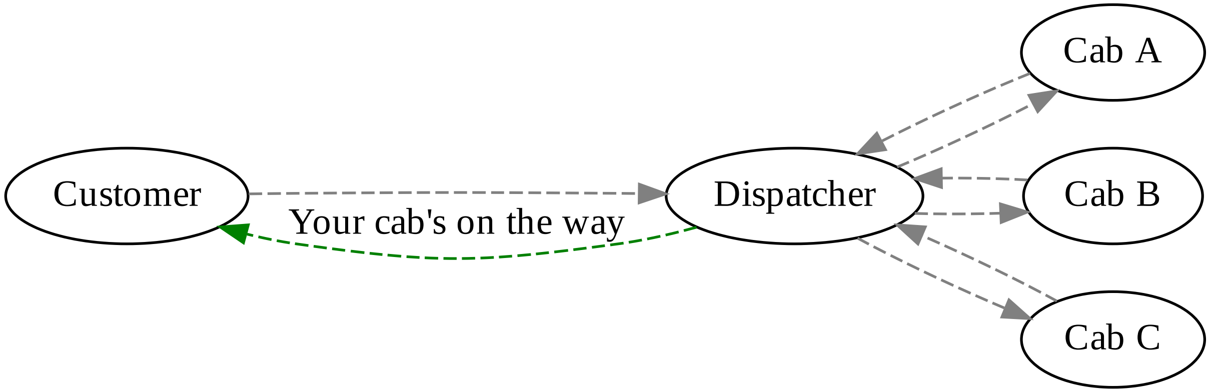
rusty-cabs

Flow



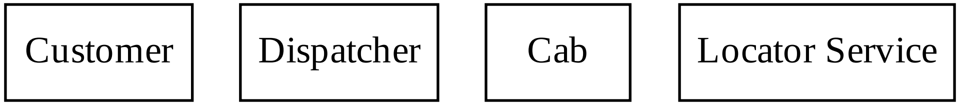
rusty-cabs

Flow



rusty-cabs

Design



Customer

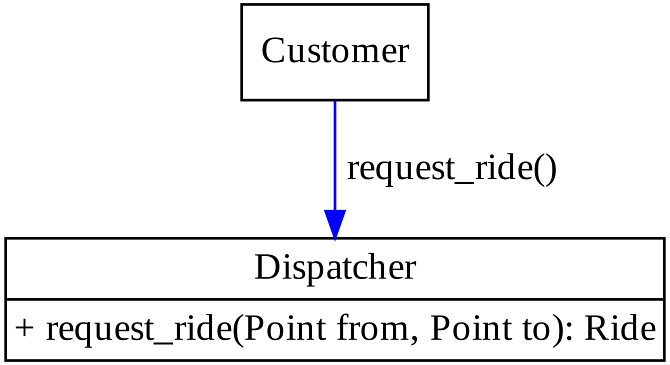
Dispatcher

Cab

Locator Service

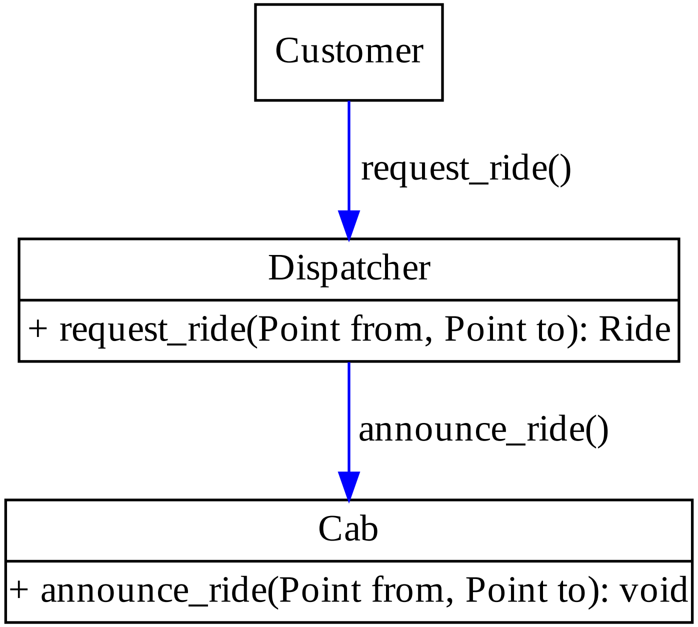
rusty-cabs

Design



rusty-cabs

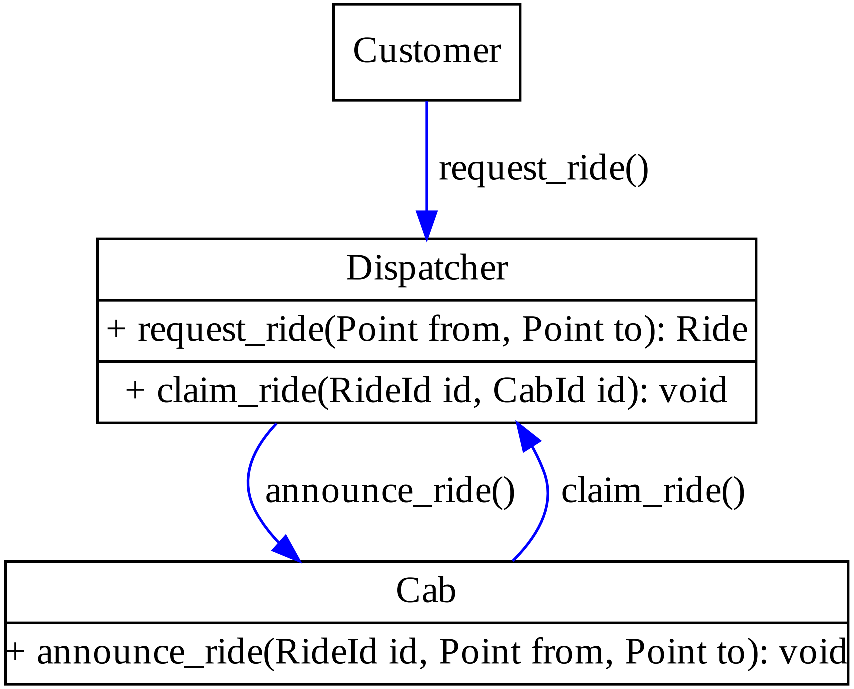
Design



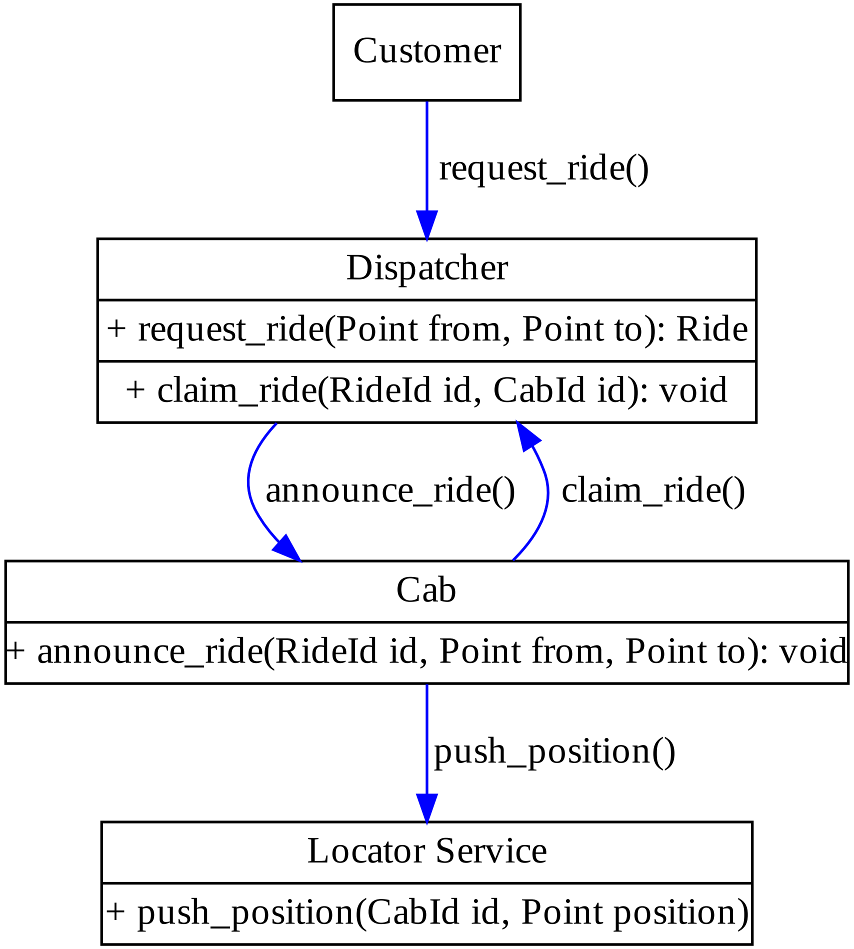
Customer
request_ride()
Dispatcher
+ request_ride(Point from, Point to): Ride
announce_ride()
Cab
+ announce_ride(Point from, Point to): void

rusty-cabs

Design


Customer
request_ride()
Dispatcher
+ request_ride(Point from, Point to): Ride
+ claim_ride(RideId id, CabId id): void
announce_ride()
claim_ride()
Cab
+ announce_ride(RideId id, Point from, Point to): void

rusty-cabs


Customer
request_ride()
Dispatcher
+ request_ride(Point from, Point to): Ride
+ claim_ride(RideId id, CabId id): void
announce_ride()
claim_ride()
Cab
+ announce_ride(RideId id, Point from, Point to): void
push_position()
Locator Service
+ push_position(CabId id, Point position)

rusty-cabs

Let’s see how it actually end up like!

Summary

I lied

Summary

I lied

At the beginning, I said that actors are those message-acting objects...

Summary

I lied

At the beginning, I said that actors are those message-acting objects…

… but that's actually a *gross* oversimplification.

Summary

I lied

At the beginning, I said that actors are those message-acting objects…

… but that’s actually a *gross* oversimplification.

Actor model originated in **1973** and lots of things have happened since then.

Summary

I understated

Table 2. Conveniently passed in silence

Summary

I understated

Table 3. Conveniently passed in silence

Supervision / Fault tolerance	What happens when an actor dies?

Summary

I understated

Table 4. Conveniently passed in silence

Supervision / Fault tolerance	What happens when an actor dies?
Persistence	What happens when the application’s restarted?

Summary

I understated

Table 5. Conveniently passed in silence

Supervision / Fault tolerance	What happens when an actor dies?
Persistence	What happens when the application’s restarted?
Network	What happens if we want our application to be distributed?

Summary

I understated

And quite a bit more:

- actor discovery,
- cancellable messages,
- event buses,
- event sourcing,
- transactions,
- (...)

Summary

I understated

There are comprehensive actor frameworks (like **Akka**) that solve practically all of those issues - Rust, though, still has a long journey ahead of it.

We have a few crates to select from - namely: **Actix**, **Bastion** or **Riker** - but for now they aren't nearly as half as complete as the ones known in other languages.

Summary

What problems do actors introduce?

Summary

What problems do actors introduce?

Code bloat, increased **complexity** or overzealous **Arc-ing** are some of the obvious issues that may arise, but let’s focus on a different aspect - a more language-agnostic one.

Summary

What problems do actors introduce?

Let’s state a few facts:

Summary

What problems do actors introduce?

Let’s state a few facts:

1. Actors are self-contained workers.

Summary

What problems do actors introduce?

Let’s state a few facts:

1. Actors are self-contained workers.
2. You don’t invoke actors' code directly, you send messages and patiently wait.

Summary

What problems do actors introduce?

Let’s state a few facts:

1. Actors are self-contained workers.
2. You don’t invoke actors' code directly, you send messages and patiently wait.
3. They work miracles when scattered across many servers.

Summary

What problems do actors introduce?

Let’s state a few facts:

1. Actors are self-contained workers.
2. You don’t invoke actors' code directly, you send messages and patiently wait.
3. They work miracles when scattered across many servers.

... so actors are actually your typical kafka-esque / rabbitmq-esque microservices!

Summary

What problems do *distributed systems* introduce?

Summary

What problems do *distributed systems* introduce?

1. Testing distributed systems is *moderately hard*.

Summary

What problems do *distributed systems* introduce?

1. Testing distributed systems is *moderately hard*.
2. Handling Byzantine faults is *hard*.

Summary

What problems do *distributed systems* introduce?

1. Testing distributed systems is *moderately hard*.
2. Handling Byzantine faults is *hard*.
3. Handling transactions spanning across many different sub-systems is *really hard*.

Summary

What problems do *distributed systems* introduce?

1. Testing distributed systems is *moderately hard*.
2. Handling Byzantine faults is *hard*.
3. Handling transactions spanning across many different sub-systems is *really hard*.
4. Debugging tons of tiny actors over network is *impossibly hard*.

Summary

What problems do *distributed systems* introduce?

1. Testing distributed systems is *moderately hard*.
2. Handling Byzantine faults is *hard*.
3. Handling transactions spanning across many different sub-systems is *really hard*.
4. Debugging tons of tiny actors over network is *impossibly hard*.
5. Keeping actors' protocols in sync *can be hard*.

Summary

When should you use actor model then?

Summary

When should you use actor model then?

Honestly, it depends...

Fantastic Actors and Where to Find Them

~ Patryk Wychowaniec, 2020

Thank you!