DDD, Event sourcing and CQRS

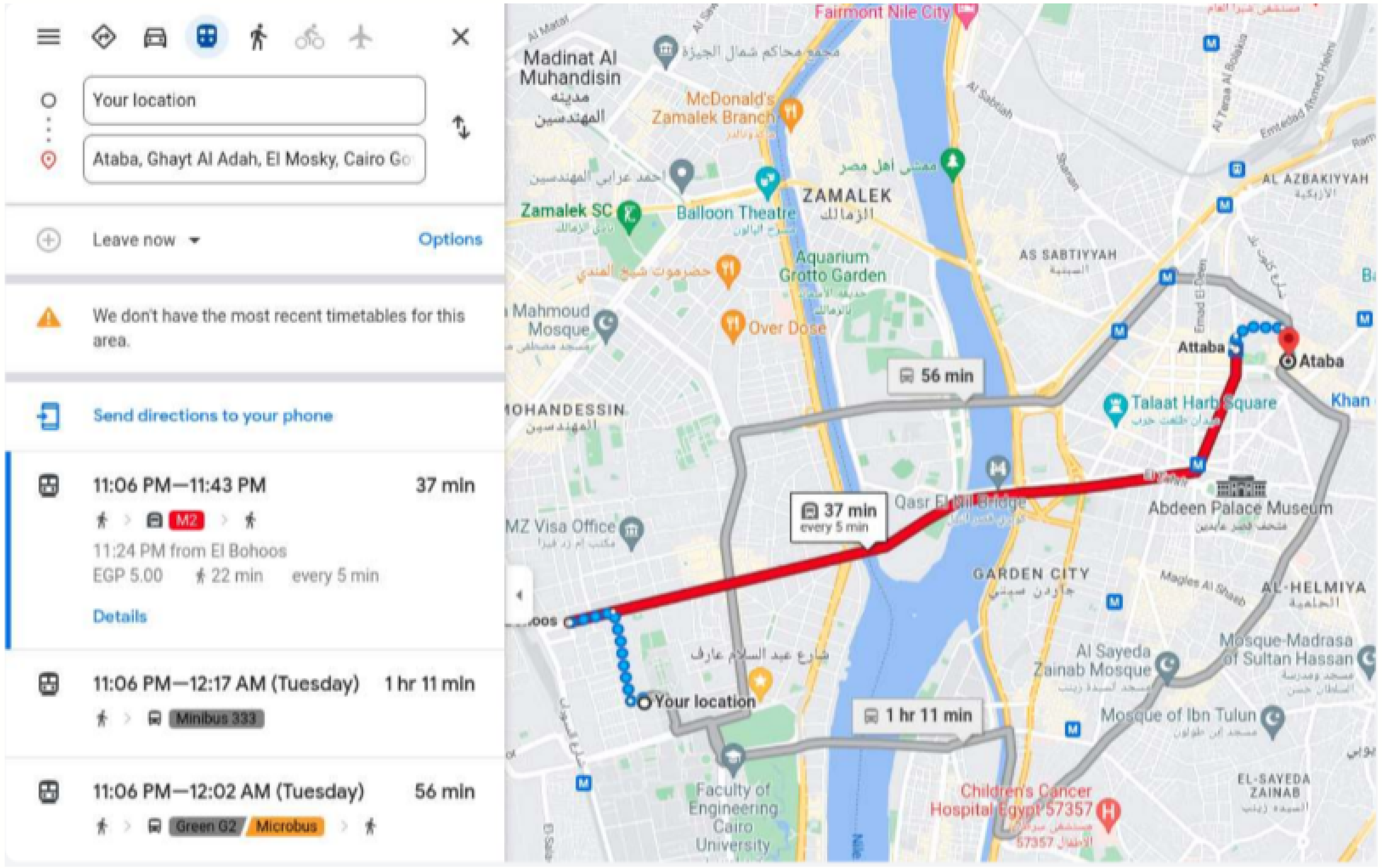
A Practical Guide

A little bit about us.

Transport for Cairo

Transport for Cairo مواصلة للقاهرة

We put microbuses in google maps



Your location

Ataba, Ghayt Al Adah, El Mosky, Cairo Go

Leave now ▾ Options

We don't have the most recent timetables for this area.

Send directions to your phone

11:06 PM—11:43 PM 37 min

M2

11:24 PM from El Bohoos

EGP 5.00 22 min every 5 min

Details

11:06 PM—12:17 AM (Tuesday) 1 hr 11 min

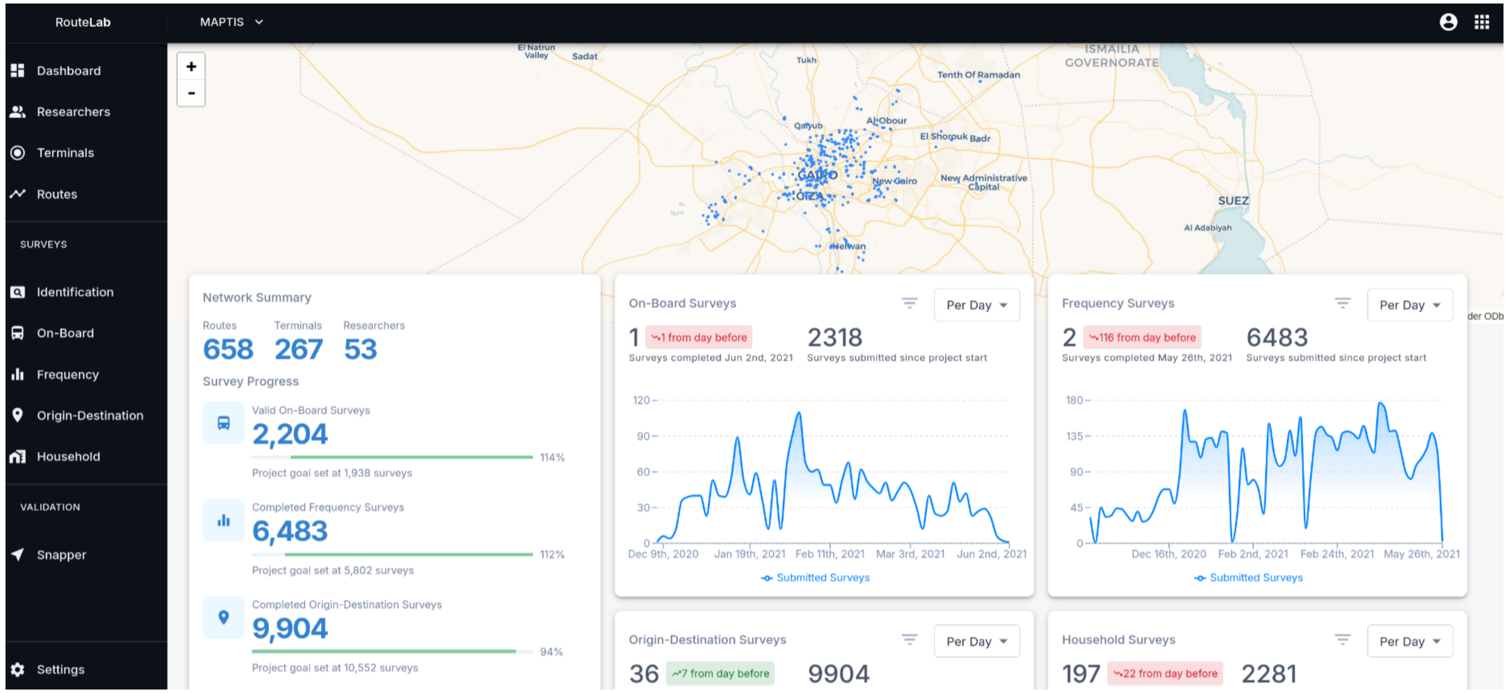
Minibus 333

11:06 PM—12:02 AM (Tuesday) 56 min

Green G2 / Microbus

RouteLab

- Transit Data Collection and analysis **at scale**
- Models the complexity of informal transit
- Used in 10+ african cities



Network Summary

Routes 658 · Terminals 267 · Researchers 53

Survey Progress

Valid On-Board Surveys
2,204 — 114%
Project goal set at 1,938 surveys

Completed Frequency Surveys
6,483 — 112%
Project goal set at 5,802 surveys

Completed Origin-Destination Surveys
9,904 — 94%
Project goal set at 10,552 surveys

On-Board Surveys
1 — 1 from day before — Surveys completed Jun 2nd, 2021
2318 — Surveys submitted since project start

Frequency Surveys
2 — 116 from day before — Surveys completed May 26th, 2021
6483 — Surveys submitted since project start

Origin-Destination Surveys
36 — 7 from day before
9904

Household Surveys
197 — 22 from day before
2281

From DB-centric design to Event Sourcing

- We used to keep most of the state on the server
- Increasing need for asynchronicity (due to field work conditions)

Overview of DDD with Functional Event Sourcing

So What is DDD basically?

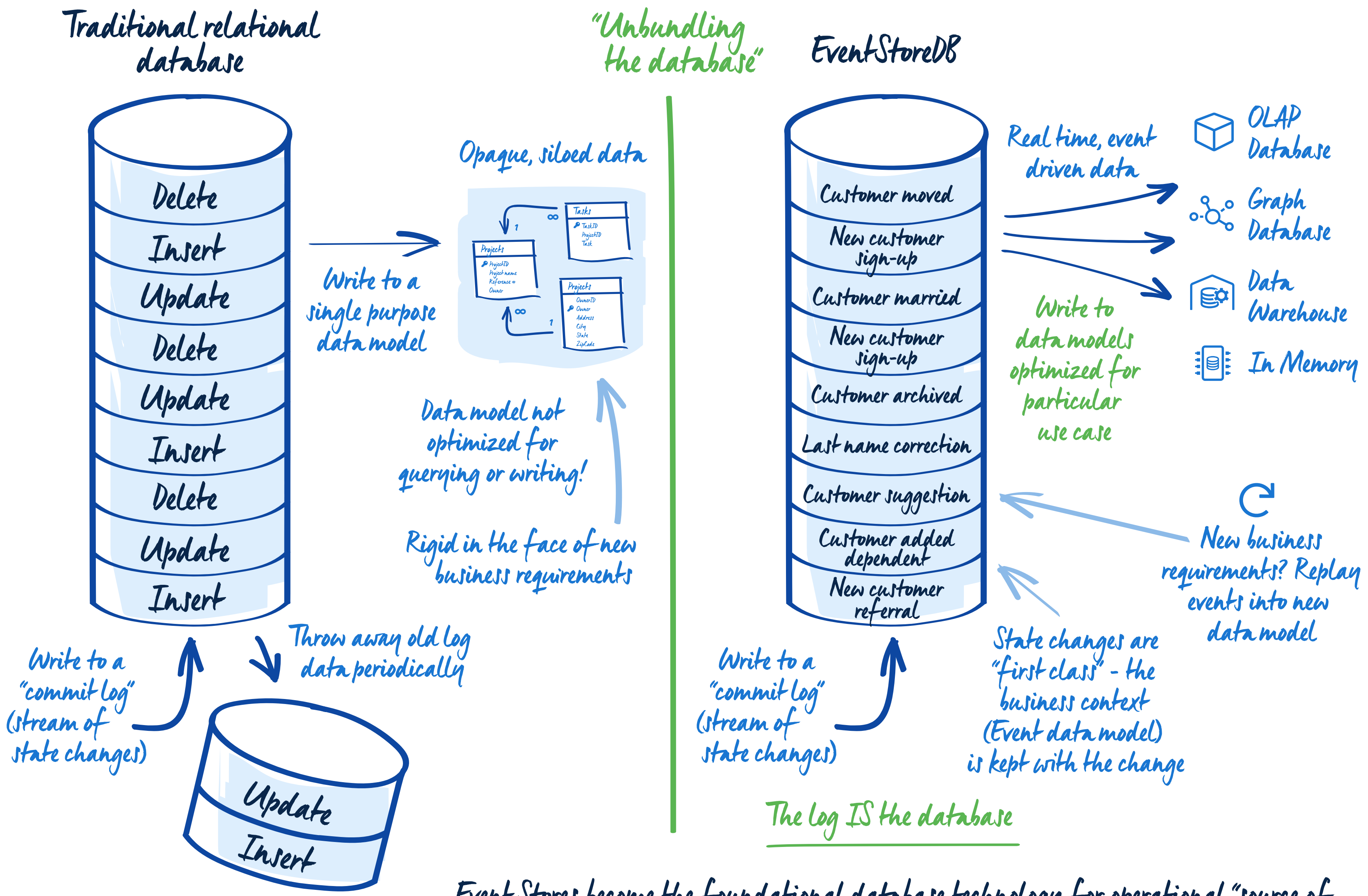
Align software design closely with the business domain. The goal is to ensure that the code reflects the actual business processes, rules, and language.

What does software do?

(mostly...)

It's about modeling reality.
a sequence of immutable facts.

Event Stores represent an unbundling of traditional database technology into a more powerful and flexible alternative


Traditional relational database
Delete
Insert
Update
Delete
Update
Insert
Delete
Update
Insert
Write to a single purpose data model
Opaque, siloed data
Tasks
Projects
Projects
Data model not optimized for querying or writing!
Rigid in the face of new business requirements
Write to a "commit log" (stream of state changes)
Throw away old log data periodically
Update
Insert
"Unbundling the database"
EventStoreDB
Customer moved
New customer sign-up
Customer married
New customer sign-up
Customer archived
Last name correction
Customer suggestion
Customer added dependent
New customer referral
Real time, event driven data
OLAP Database
Graph Database
Data Warehouse
In Memory
Write to data models optimized for particular use case
New business requirements? Replay events into new data model
State changes are "first class" - the business context (Event data model) is kept with the change
Write to a "commit log" (stream of state changes)
The log IS the database

Event Stores become the foundational database technology for operational "source of truth" data. Other "made for purpose" databases remain relevant, including relational, but layer over event streams to be used for specific query and analytics use cases.

Why should I care

- Naturally aligns itself with the domain.
- Requirement evolution is a breeze (🤫 no db migrations)
- No drift between log output and code.
- Pure,replayable,reliable,safe,*accident* proof.

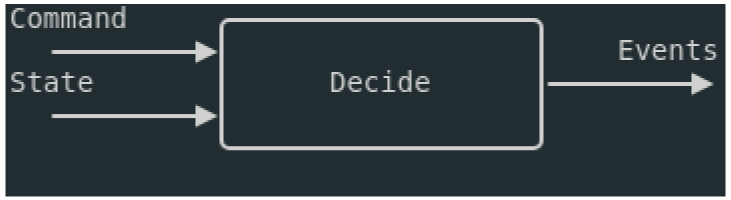
And performant

Our recent project.

300k events/day. With a single small pod

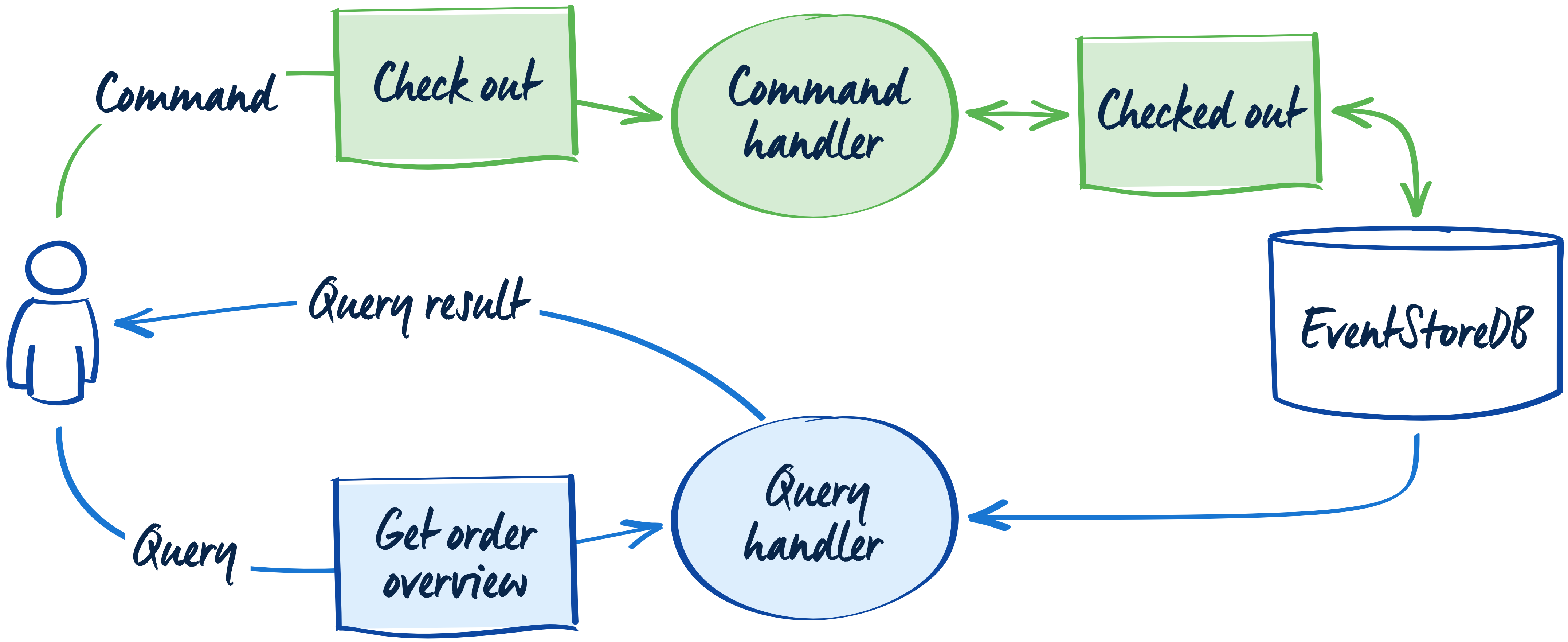
The basic idea:

- Commands are intentions.
- Events are facts.



“ Source: Functional decider by Jeremie Chassaing ”

Write model and read model



(The write side): Evolution function and The decider

- Decider looks at the command and the given system state and decides what the next "fact(s)" are going to be.
- For each "entity/service" you are familiar with "service->repository->database"
- Here we define an "aggregate" Each aggregate has it's evolution function that determines it's state from previous events.
- Right now you can think of an aggregate as an entity.

In practice: processing commands into new facts.

```
// streamId is entityPrefix-<randomId>
const { events } = await eventStore.readStream(streamId);
const state = events.reduce(evolve, { status: "empty" });

// Either type Left | Right. Success Or failure
const decideResult = decide(command, state);
if (isLeft(decideResult)) {
  return unwrapEither(deciderResult);
}

const resultEither = await eventStore.appendToStream(
  streamId,
  unwrapEither(decideResult)
);
```

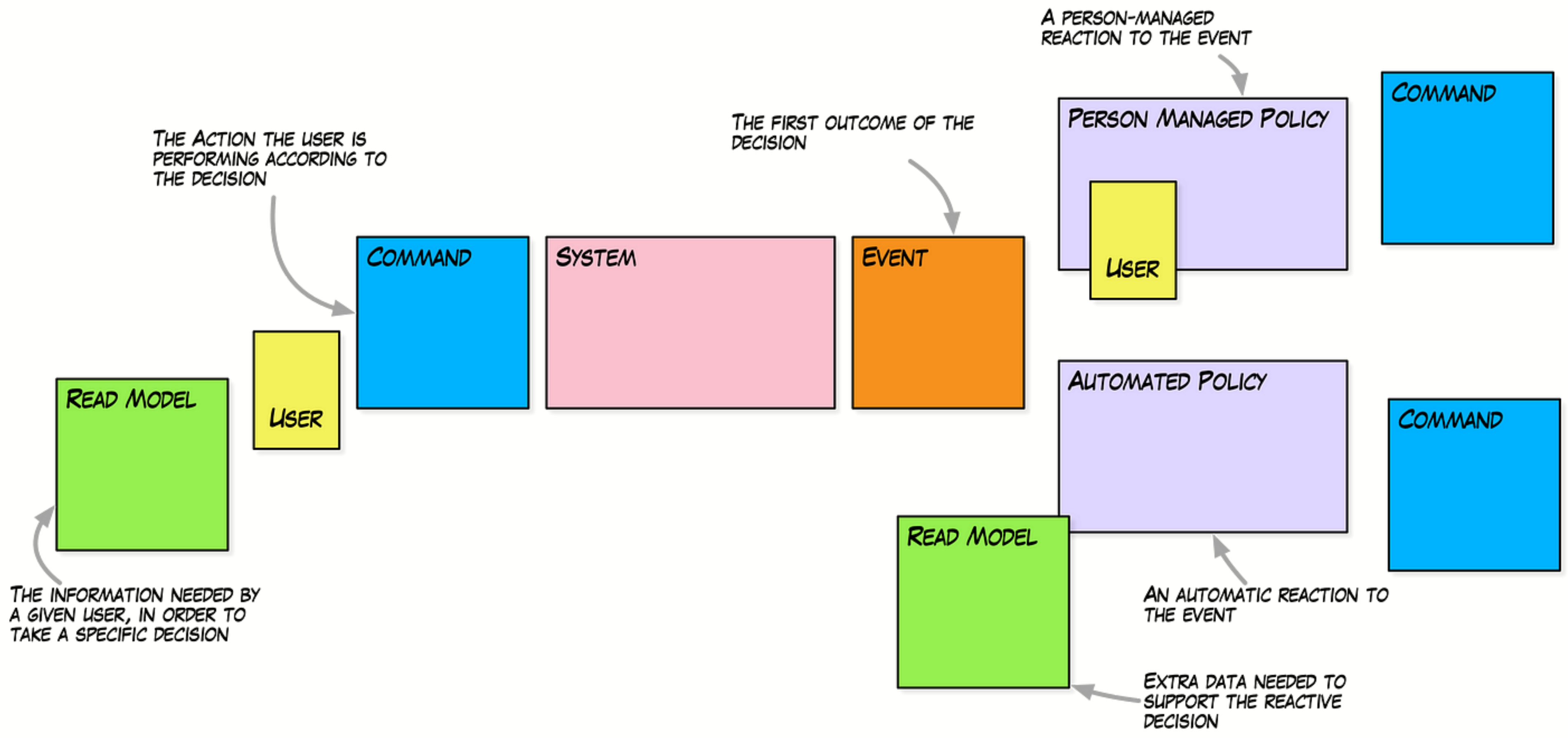
decide ex:1

```
match(state,command).with(
  {
    command: { type: "AssignedFrsAddFr" },
    state: { status: "planned" },
  },
  ({ command: { data } }) =>
    makeRight({
      type: "TrafficCounts.Survey.AssignedFRs.FRAdded",
      data,
    }),
)
```

decide ex:2

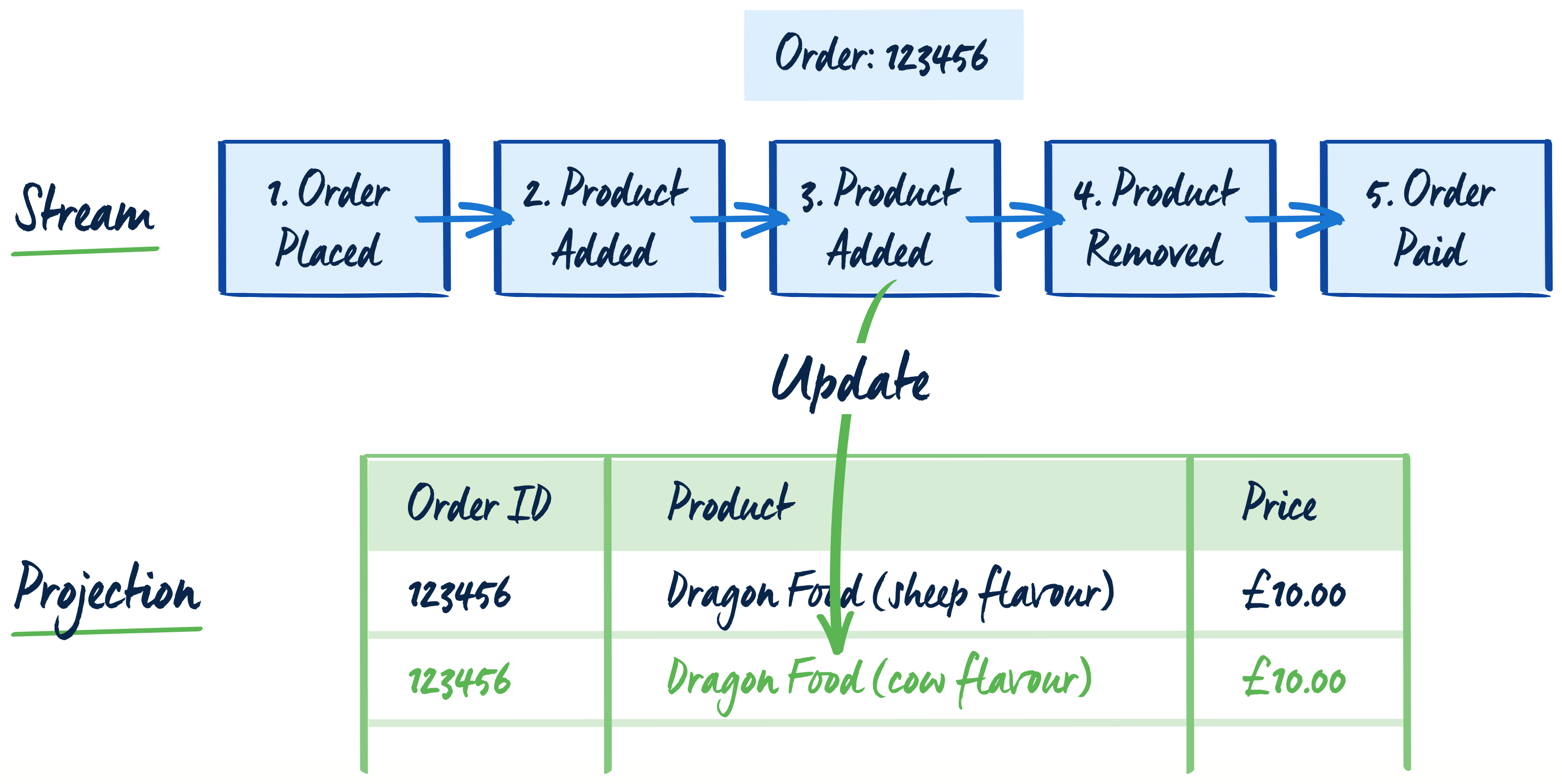
```
match(state,command).with(
  {
    command: {
      type: "ReserveSegmentModePair",
    },
    state: {
      status: P.union("planned", "inProgress"),
    },
  },
  ({ command: { data }, state }) =>
    match(state.segmentModePairsReservations)
      .with(
        { [data.segmentId + data.modeId]: { available: false } },
        () =>
          makeLeft({ type: "ALREADY_RESERVED" as "ALREADY_RESERVED" }),
      )
      .otherwise(() =>
        makeRight({
          type: "TrafficCounts.Survey.SegmentModePairReservations.PairReserved",
          data,
        }),
      )),
```

Summary Diagram


A PERSON-MANAGED REACTION TO THE EVENT
THE ACTION THE USER IS PERFORMING ACCORDING TO THE DECISION
THE FIRST OUTCOME OF THE DECISION
PERSON MANAGED POLICY
COMMAND
USER
COMMAND
SYSTEM
EVENT
READ MODEL
USER
AUTOMATED POLICY
COMMAND
READ MODEL
THE INFORMATION NEEDED BY A GIVEN USER, IN ORDER TO TAKE A SPECIFIC DECISION
AN AUTOMATIC REACTION TO THE EVENT
EXTRA DATA NEEDED TO SUPPORT THE REACTIVE DECISION

“ Source: Alberto Brandolini ”

Projections



Order ID	Product	Price
123456	Dragon Food (sheep flavour)	£10.00
123456	Dragon Food (cow flavour)	£10.00

“ Source: Event Store DB Blog ”

Projections 2

They can be in-memory.
Use the correct data structure/database for *your* problem.

No 'Database' per service. Single source of truth

No relational schema means no need for service data boundaries/ownership. Reduce service interdependence for read operations.

Eventual consistency

- The read model technically lags behind the write model
- Solution: The API only accepts commands when the client sends the latest event-id within the aggregate’s stream

Other concerns for a production system

- Event versioning and upgrades.
- Easily spin up different views for different needs.
- Live projections (🤔)
- Efficiently projecting a large event stream.

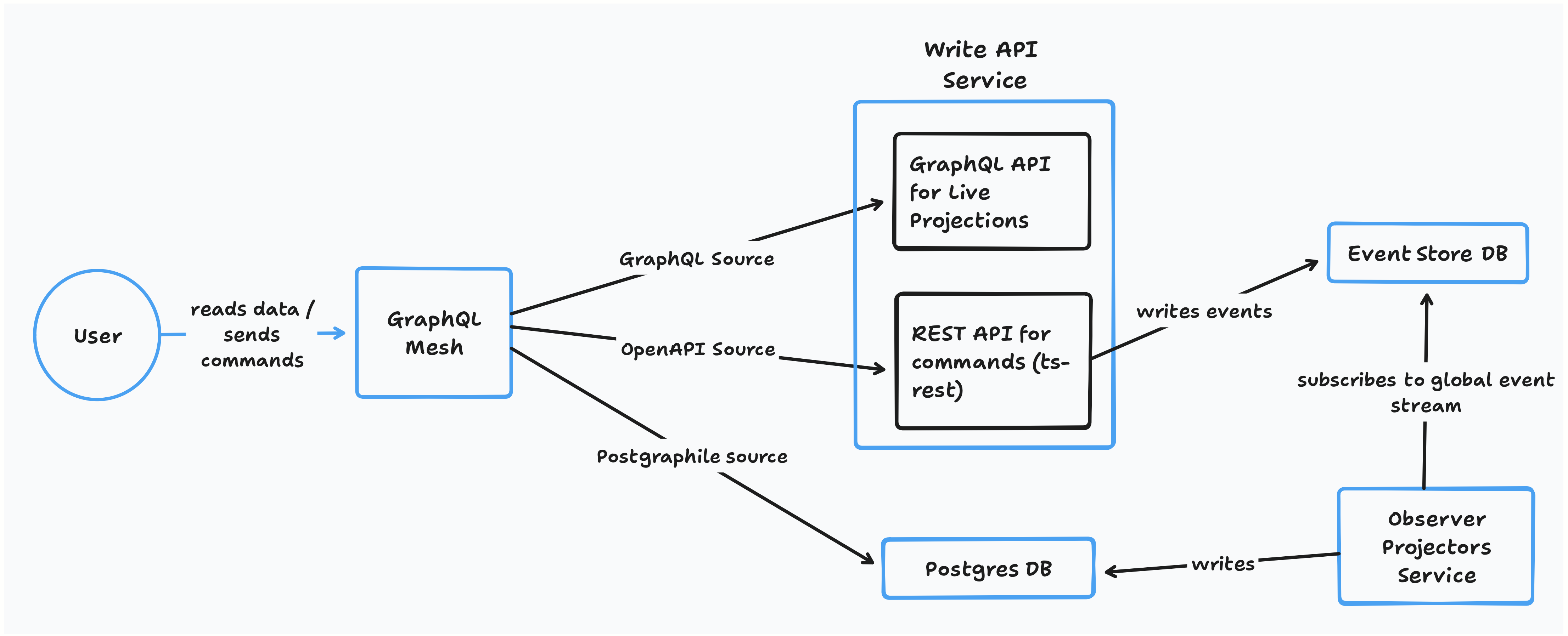
Lessons learned from our implementation

- Zod
- Typescript
- TONS of Codegen
- Graphql
- Postgraphile.
- ts-rest
- ts-pattern

Small Example Project

- A very simple aggregate
- Demonstrates the usage of said technologies together
- GitHub repo link: https://github.com/hzmmohamed/ddd-es-example

Our System's Architecture


Write API Service
GraphQL API for Live Projections
REST API for commands (ts-rest)
Event Store DB
User
reads data / sends commands
GraphQL Mesh
GraphQL Source
OpenAPI Source
Postgraphile source
writes events
subscribes to global event stream
Postgres DB
writes
Observer Projectors Service

Our System's Architecture

- Read and write API separation
- One GraphQL API gateway
- *TypeScript types + code-gen*
  A fully type-safe system from the domain model to the GraphQL API client on the front-end

Further Resources

Invaluable resources for learning event sourcing:

- Oskar Dudycz's Blog
- The Functional Decider Pattern
- https://www.eventstore.com/event-sourcing
- Greg Young's Book on Event Versioning
- Greg Young's talks, generally

Thank you for listening.

TfC is hiring!

- h.fhami@transportforcairo.com
- emad.moh.hamdy@gmail.com

You can find today's talk slides on GitHub: https://github.com/Mohamedemad4/ddd-es-presentation-slides