How slow is your tram? 🚊

using STTP, CE3, FS2 and scala-cli

Photo by Guillaume.G CC BY-SA 4.0

Great stories feature

Great villains

Star Wars

Batman

And...

My story

My name is Michał 👋

- Senior Software engineer @ SiriusXM
- Blog about Scala
- OSS from time to time

My name is Michał 👋

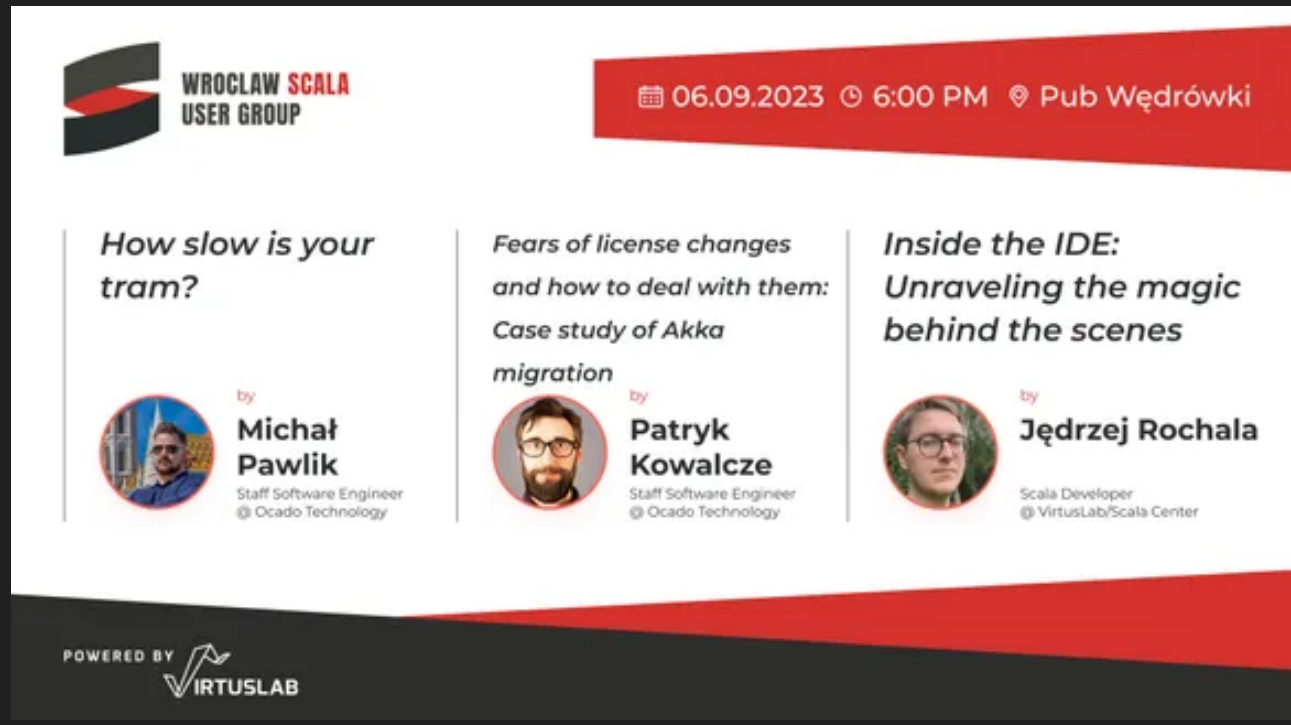
I have bad luck with public transportation

My name is Michał 👋

I have bad luck with public transportation

And this is my fs2 story with trams

Like when I was giving this talk in Wrocław



They hid the entrance to the venue

Or the other day

They took a bus stop too literally

Foto: Mateusz Iwanczyk Photo

This happens to me

Foto: Jarosław Jakubczak

from time

Foto: wroclawskakomunikacja.pl

to time

Foto: Paweł Relikowski / Polska Press

Thus I asked myself

How slow is your tram?

Let's try to find out

Plan

1. Find data source of vehicle positions
2. Fetch and parse
3. Fetch some more
4. Calculate diffs
5. Produce statistics

Find data source of vehicle positions

Interactive map

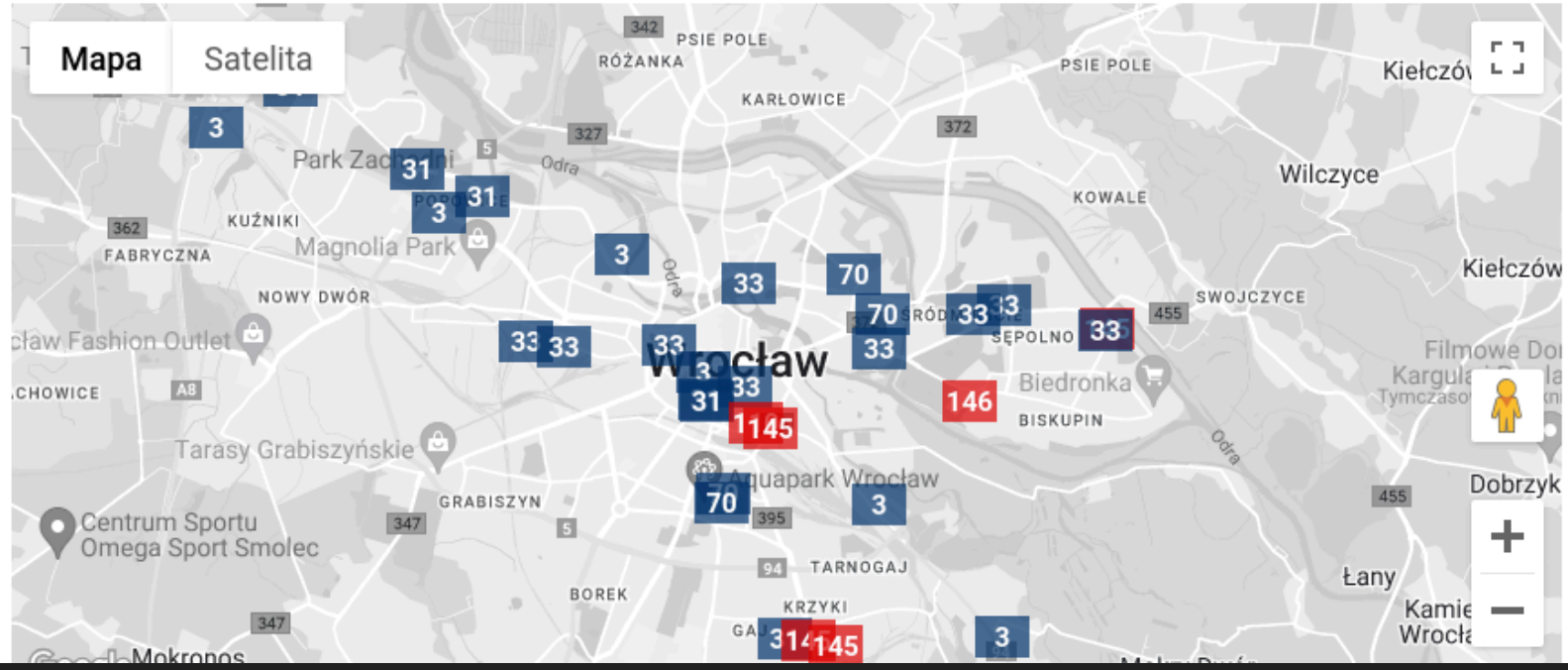
https://mpk.wroc.pl/strefa-pasazera/zaplanuj-podroz/mapa-pozycji-pojazdow

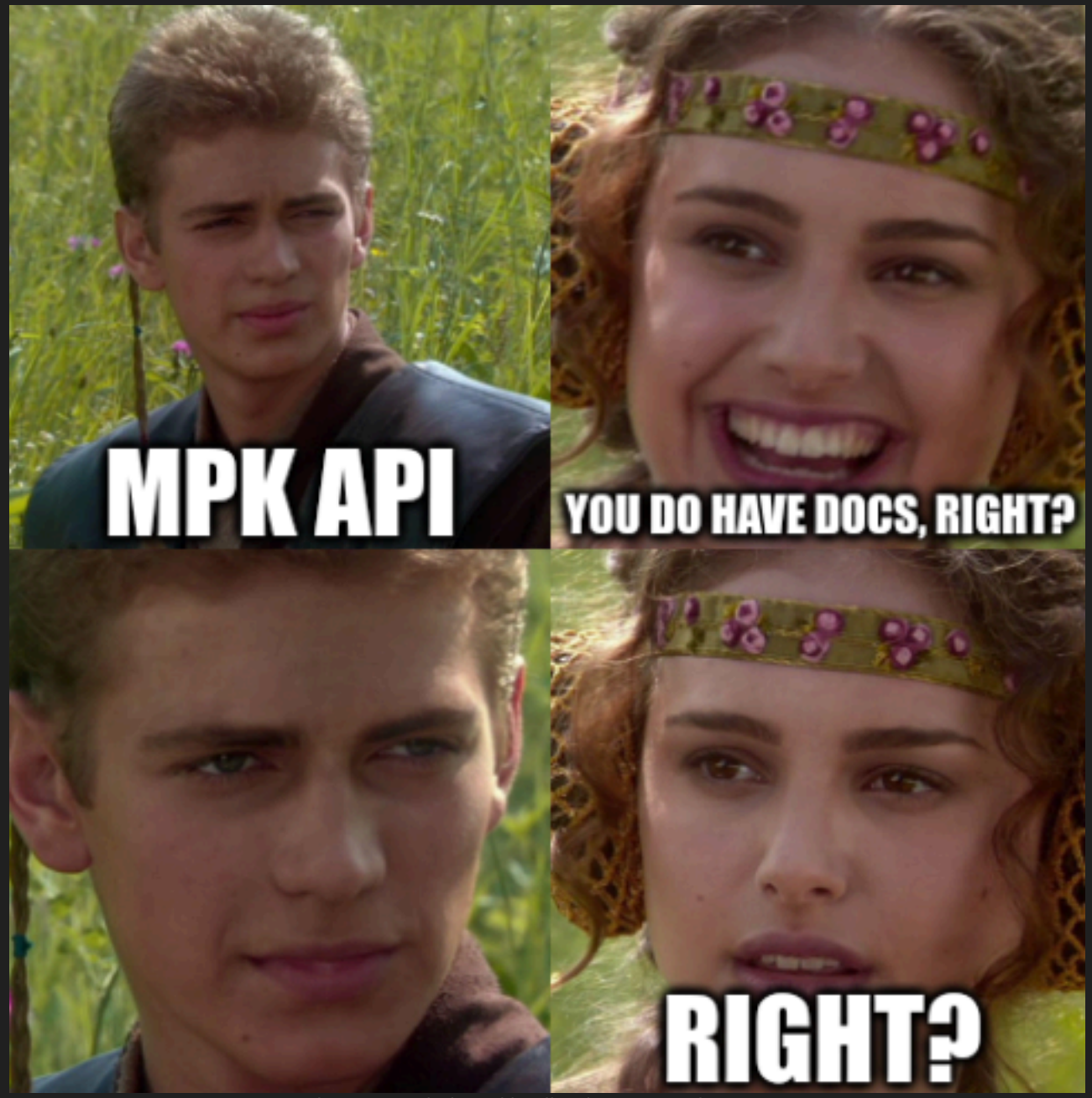
Autobusy

A	D	K	N	100	101	102	103	104
105	106	107	108	109	110	111	112	113
114	115	116	118	119	120	121	122	124
125	126	127	128	129	130	131	132	133
134	136	142	143	144	145	146	147	148
149	150	151	206	240	241	242	243	244
245	246	247	248	249	250	251	253	255
257	259	315	319	602	607	714	733	

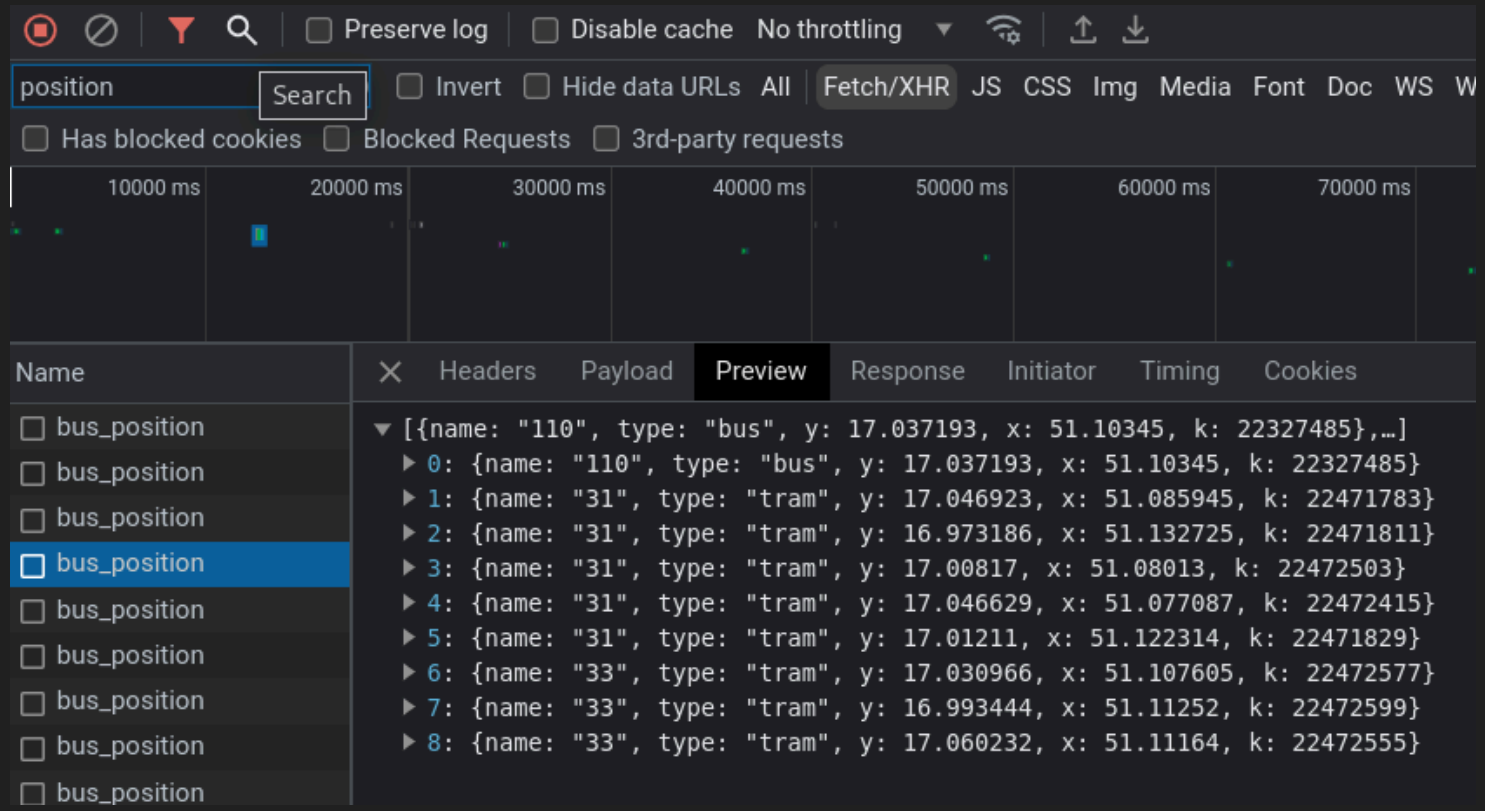
Tramwaje

1	2	3	4	5	6	7	8	9
10	11	15	16	17	18	19	20	23
31	33	70	74					





Inspect 🕵



Investigate 🕵

```
curl -s 'https://mpk.wroc.pl/bus_position' \
  -H 'accept: application/json, text/javascript, */*; q=0.01' \
  -H 'content-type: application/x-www-form-urlencoded; charset=UTF-8' \
  --data-raw 'busList%5Bbus%5D%5B%5D=110&busList%5Btram%5D%5B%5D=31&busList%5Btram%5D%5B%5D=33' \
  --compressed | jq
[
  {
    "name": "31",
    "type": "tram",
    "y": 17.051546,
    "x": 51.076923,
    "k": 22471783
  },
  {
    "name": "31",
    "type": "tram",
    "y": 17.049835,
    "x": 51.081802,
    "k": 22472415
  },
 /*  ...  */
]
```

Finally data!

```
[
  {
    "name": "31",
    "type": "tram",
    "y": 17.051546,
    "x": 51.076923,
    "k": 22471783
  },
 /*  ...  */
]
```

What does it mean? 🤔

- `name` - line name like `31`, `33`, `110`
- `type` - one of `tram`, `bus`
- `y` - latitude
- `x` - longitude
- `k` - 🤨

What does it mean? 🤔

- `name` - line name like `31`, `33`, `110`
- `type` - one of `tram`, `bus`
- `y` - latitude
- `x` - longitude
- `k` - 🤨 looks like a vehicle id

Plan

1. Find data source of vehicle positions ✅
2. Fetch and parse
3. Fetch some more
4. Calculate diffs
5. Produce statistics

Coding time 💻

Fetch and parse

Shape of our request

- HTTP POST
- List of vehicles like `busList[bus][]=110&busList[tram][]=31&busList[tram][]=33`
- `content-type: application/x-www-form-urlencoded`
- Expect JSON output

STTP Client

```scala
val apiUri = uri"https://mpk.wroc.pl/bus_position"

def payload(buses: List[String], trams: List[String]) =
  (trams.map(v ⇒ s"busList[tram][]=$v") ++
    buses.map(v ⇒ s"busList[bus][]=$v")).mkString("&")

// ☝ builds this thing: busList[bus][]=110&busList[tram][]=31&busList[tram][]=33
```

STTP

Data model

```
1. case class Record(
2.   name: String,
3.   x: Double,
4.   y: Double,
5.   k: Int // 😐
6. ) derives Codec.AsObject // This will generate JSON Encoder and Decoder
```

STTP

```
1. def request(
2.     backend: SttpBackend[IO, Any], buses: List[String], trams: List[String]
3.   ): IO[List[Record]] = // Note the return type
4.   basicRequest
5.     .post(apiUri)
6.     .body(payload(buses, trams)) // Something like busList[bus][]=110&busList[tram][]=31&busList[tram][]=33
7.     .contentType(MediaType.ApplicationXWwwFormUrlencoded)
8.     .response(asJson[List[Record]])
9.     .send(backend)
10.    .map(_.body)                 // We are only interested in the result
11.    .rethrow                     // Fail `IO` on all errors, we are being simple here
```

Let's run it

- Our `request` returns an `IO`, so we need a way to execute it
- It requires `SttpBackend` so we need to create one

Let's run it

The easiest way to execute an `IO` is to create a `Main` class that handles it for us

```
object Main extends IOApp.Simple {
  def run: IO[Unit] = ??? // our logic goes here
}
```

Let's run it

Let's create a backend, execute the request and print the result

```
1. object Main extends IOApp.Simple {
2.   val buses = List("110")
3.   val trams = List("31", "33")
4.
5.   def run =
6.     HttpClientFs2Backend
7.       .resource[IO]()
8.       .use(backend ⇒ request(backend, buses, trams))
9.       .flatMap(IO.println)
10. }
```

Let's run it

```scala
object Main extends IOApp.Simple {
  val buses = List("110")
  val trams = List("31", "33")

  def run =
    HttpClientFs2Backend
      .resource[IO]()
      .use(backend ⇒ request(backend, buses, trams))
      .flatMap(IO.println)
}
```

Execution result

```
$ scala-cli sttp-client.scala

List(
  Record(31,51.141502,16.95872,22475890), Record(31,51.110912,17.02159,22475017), Record(31,51.07934,17.050734,22475050),
  Record(31,51.12252,17.011976,22475871), Record(31,51.097458,17.03275,22475942), Record(110,51.096992,17.037682,22312466),
  Record(33,51.112633,16.99349,22476133), Record(33,51.107376,17.035055,22476039), Record(33,51.11388,17.1032,22476064),
  Record(33,51.10771,17.040272,22476110)
)
```

Nice, we've got the data!

Nice, we've got the data!

but that `k: Int //` 🤨

`k: Int //` 🤨

Let's hide the API call behind an interface

```
trait Vehicles[F[_]] {
  def list(): F[Seq[Vehicle]]
}
```

- Notice the `Vehicle` type - it needs better fields than `x`, `y`, `k`

`Vehicle` model

```
1. case class Vehicle(
2.   lineName: Vehicle.LineName,
3.   measuredAt: Instant,
4.   position: Position,
5.   id: Vehicle.Id // no more `k` 🎉
6. )
```

It could be called `VehiclePosition` or `VehicleMeasurement` but let's stick with `Vehicle` for clarity

`Vehicle` model

```
1. case class Vehicle(
2.   lineName: Vehicle.LineName,
3.   measuredAt: Instant,
4.   position: Position,
5.   id: Vehicle.Id
6. )
7.
8. case class Position(latitude: Double, longitude: Double)
9.
10. object Vehicle {
11.   case class Id(value: String) extends AnyVal
12.   case class LineName(value: String) extends AnyVal
13. }
```

Refactoring time! 🚧

Our existing code

```scala
val apiUri = uri"https://mpk.wroc.pl/bus_position"

def payload(buses: List[String], trams: List[String]) =
  (trams.map(v => s"busList[tram][]=$v") ++ buses.map(v => s"busList[bus][]=$v")).mkString("&")

case class Record(
  name: String,
  x: Double,
  y: Double,
  k: Int
) derives Codec.AsObject // This will generate JSON Encoder and Decoder

def request(backend: SttpBackend[IO, Any], buses: List[String], trams: List[String]): IO[List[Record]] =
  basicRequest
    .post(apiUri)
    .body(payload(buses, trams)) // Something like busList[bus][]=110&busList[tram][]=31&busList[tram][]=33
    .contentType(MediaType.ApplicationXWwwFormUrlencoded)
    .response(asJson[List[Record]])
    .send(backend)
    .map(_.body)                 // We are only interested in the result
    .rethrow                     // Fail `IO` on all errors, we are being simple here
```

Refactoring time! 🚧

How hard can it be?

```scala
trait Vehicles[F[_]] {
  def list(): F[Seq[Vehicle]]
  // 👆 each time we call this, we receive latest data from API
}

object Vehicles {

  def mpkWrocInstance(
      backend: SttpBackend[IO, Any],
      buses: List[String],
      trams: List[String]
  ): Vehicles[IO] = ??? // make your guess
}
```

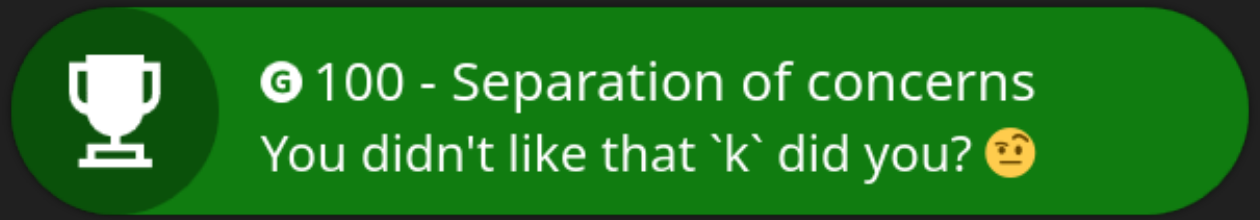
Refactoring time! 🚧

Just wrap the `request` method

```scala
trait Vehicles[F[_]] {
  def list(): F[Seq[Vehicle]]
  // 👆 each time we call this, we receive latest data from API
}

object Vehicles {

  def mpkWrocInstance(
      backend: SttpBackend[IO, Any],
      buses: List[String],
      trams: List[String]
  ): Vehicles[IO] = { () ⇒
    (request(backend, buses, trams), IO.realTimeInstant).mapN {
      (responses, now) ⇒
        responses.map { record ⇒
          Vehicle(
            lineName = Vehicle.LineName(record.name),
            measuredAt = now,
            position = Position(record.x, record.y),
            id = Vehicle.Id(record.k.toString)
          )
        }
    }
  }
}
```



`Vehicles` service done ✅

```
trait Vehicles[F[_]] {
  def list(): F[Seq[Vehicle]]
}
```

Now we have an abstract way to fetch meaningful data

```
case class Vehicle(
  lineName: Vehicle.LineName,
  measuredAt: Instant,
  position: Position,
  id: Vehicle.Id
)
```

One last thing

In the next step we'll be looking into the distance covered between two measurements

Distance

```
case class Vehicle(
  lineName: Vehicle.LineName,
  measuredAt: Instant,
  position: Position,
  id: Vehicle.Id
){
  // calculate distance in meters
  def distance(other: Vehicle): Double = ???
}
```

Distance

```
case class Vehicle(
  lineName: Vehicle.LineName,
  measuredAt: Instant,
  position: Position,
  id: Vehicle.Id
){
  // It's a shameless copy-paste from StackOverflow 😅
  def distance(other: Vehicle): Double = {
    val earthRadius = 6371000 // Earth's radius in meters
    val lat1 = toRadians(position.latitude)
    val lon1 = toRadians(position.longitude)
    val lat2 = toRadians(other.position.latitude)
    val lon2 = toRadians(other.position.longitude)
    val dlon = lon2 - lon1
    val dlat = lat2 - lat1
    val a = pow(sin(dlat / 2), 2) + cos(lat1) * cos(lat2) * pow(sin(dlon / 2), 2)
    val c = 2 * atan2(sqrt(a), sqrt(1 - a))
    val distance = earthRadius * c
    distance
  }
}
```

Distance

The point is, we can now calculate distance covered by vehicle

```
val measurement1: Vehicle = ???
val measurement2: Vehicle = ???
measurement1.distance(measurement2) // like this
```

It calculates Geographical Distance but Euclidean Distance would be good enough

Back to the plan!

Plan

1. Find data source of vehicle positions ✅
2. Fetch and parse ✅
3. Fetch some more 🛠️
4. Calculate diffs 🛠️
5. Produce statistics

Streams

Who knows what a stream is? ✋

Functional stream

Think of a sequence of data elements that

- Computes values on demand; is lazy
- Can be infinite or finite
- Can be asynchronous - supports non-blocking operations
- Has a powerful API - offers rich combinators

FS2: Functional Streams for Scala

Short intro

Let's have an exercise 🏃

- Create an infinite stream of natural numbers `1, 2, 3, 4, 5, ...`
- Keep only the odd ones `1, 3, 5, 7, ...`
- Slide through by 3 elements `(1, 3, 5), (3, 5, 7), (5, 7, 9), ...`
- Add each group `9, 15, 21, ...`
- Take first `10`

FS2: Functional Streams for Scala

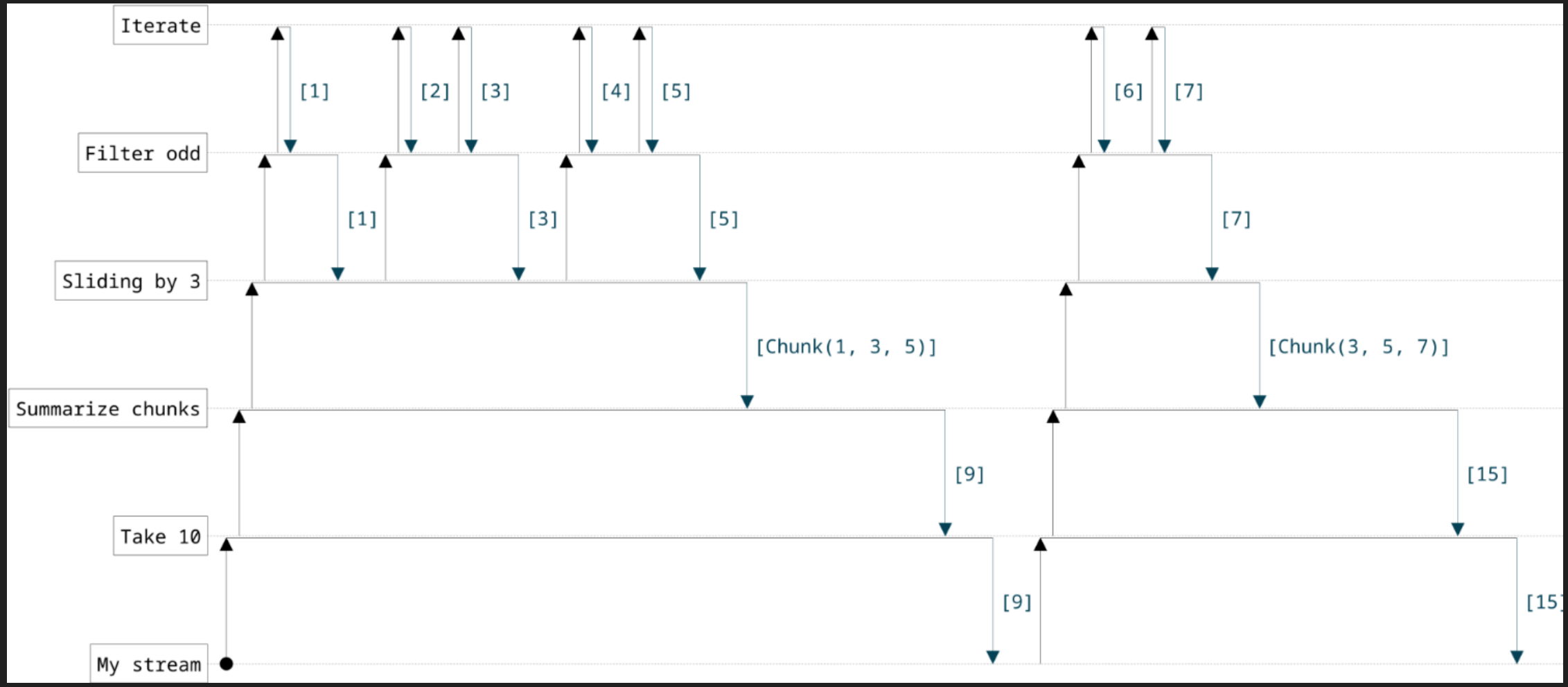
Here's how you do this

FS2: Functional Streams for Scala

```scala
Stream
  .iterate(1)(_ + 1)  // Create an infinite stream of natural numbers
  .filter(_ % 2 =!= 0) // Filter for odd numbers: 1, 3, 5, 7, 9, ...
  .sliding(3)         // Slides over each 3 elements: (1, 3, 5), (3, 5, 7), (5, 7, 9), ...
  .map { chunk =>
    chunk(0) + chunk(1) + chunk(2) // Add them together: 9, 15, 21, ...
  }
  .take(10)           // Fetch the first ten results
```

But how does it work 🤔

See it with Aquascape



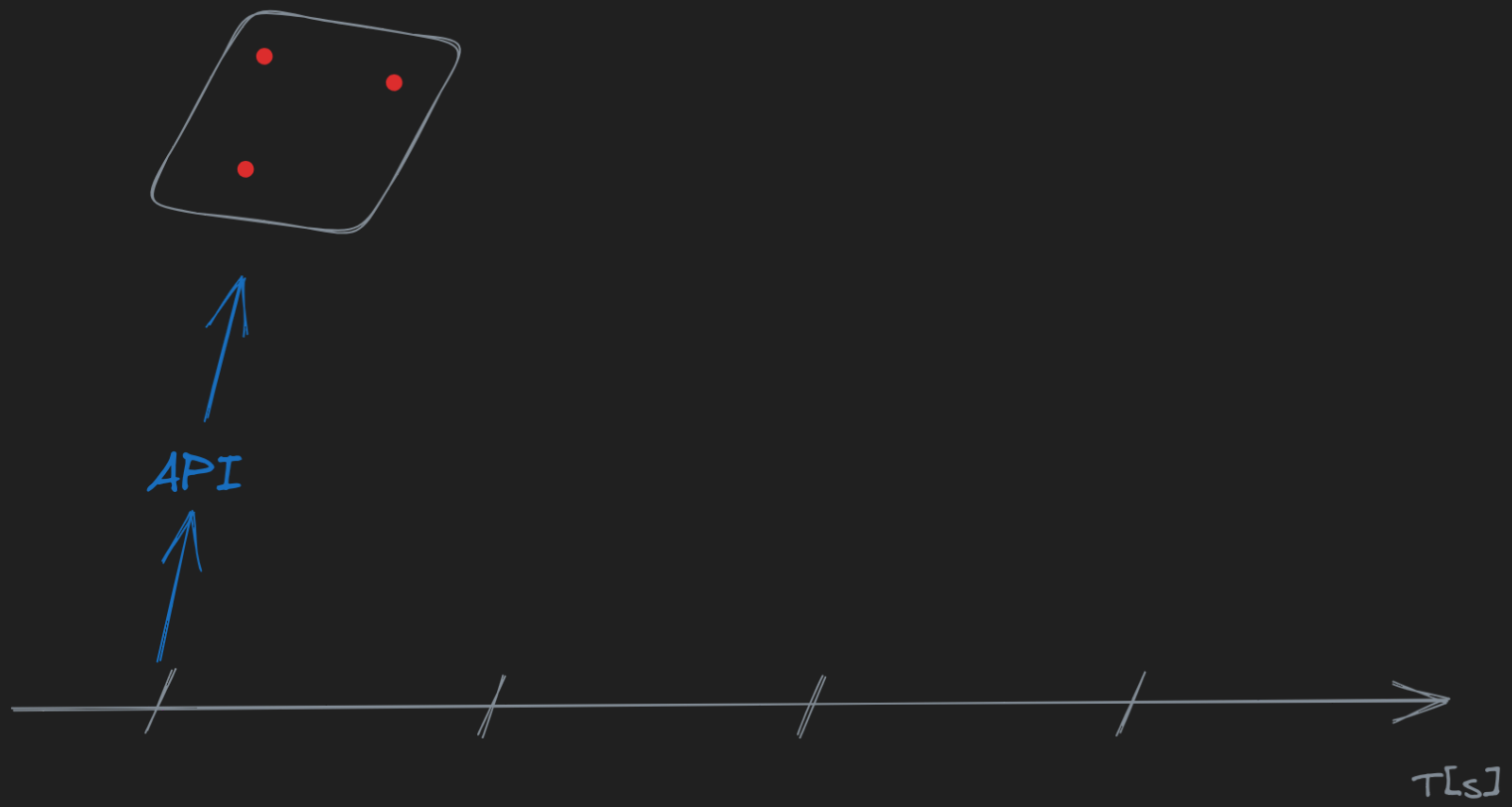
Brought to you with https://zainab-ali.github.io/aquascape

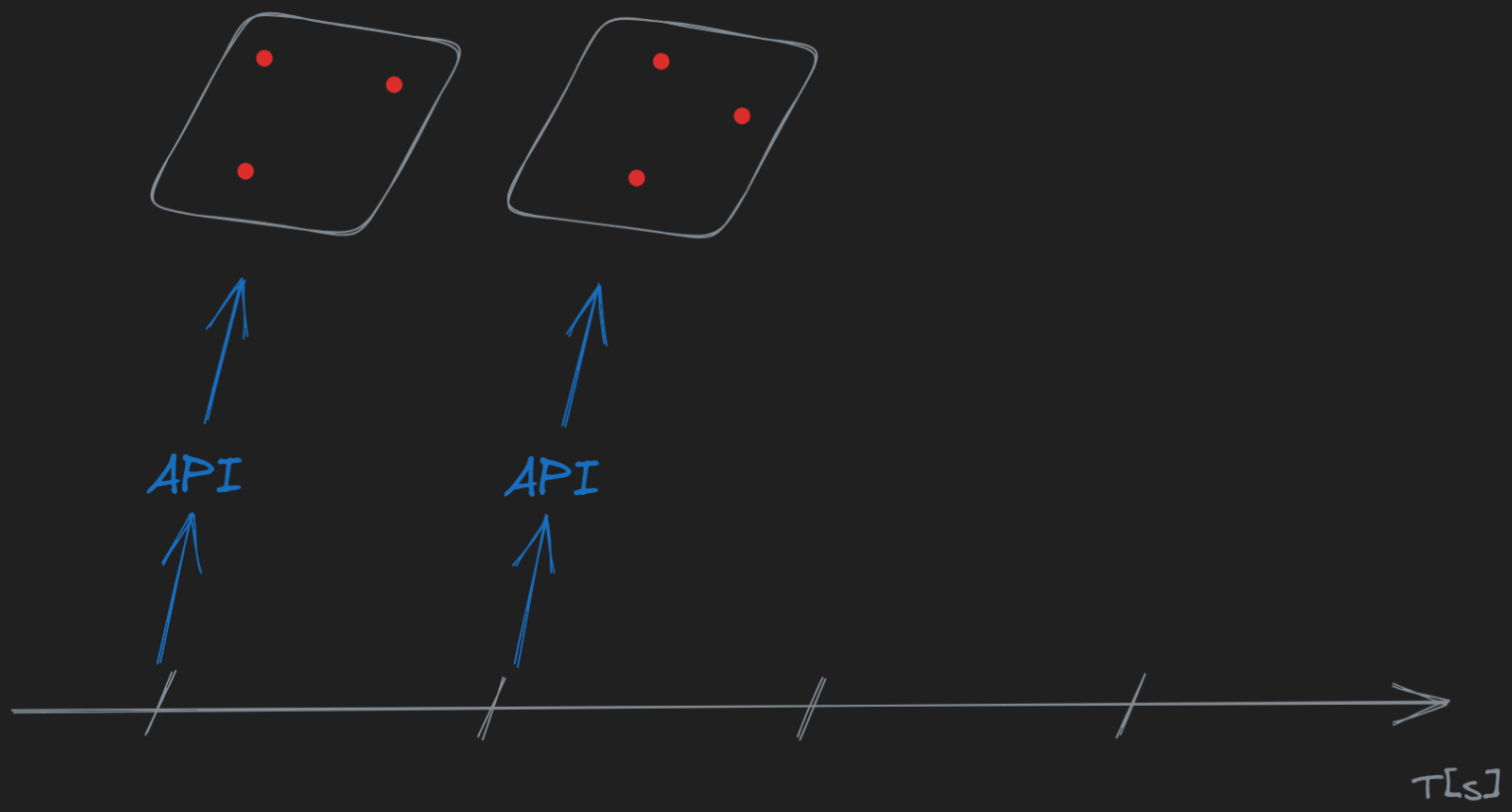
To the real use case!

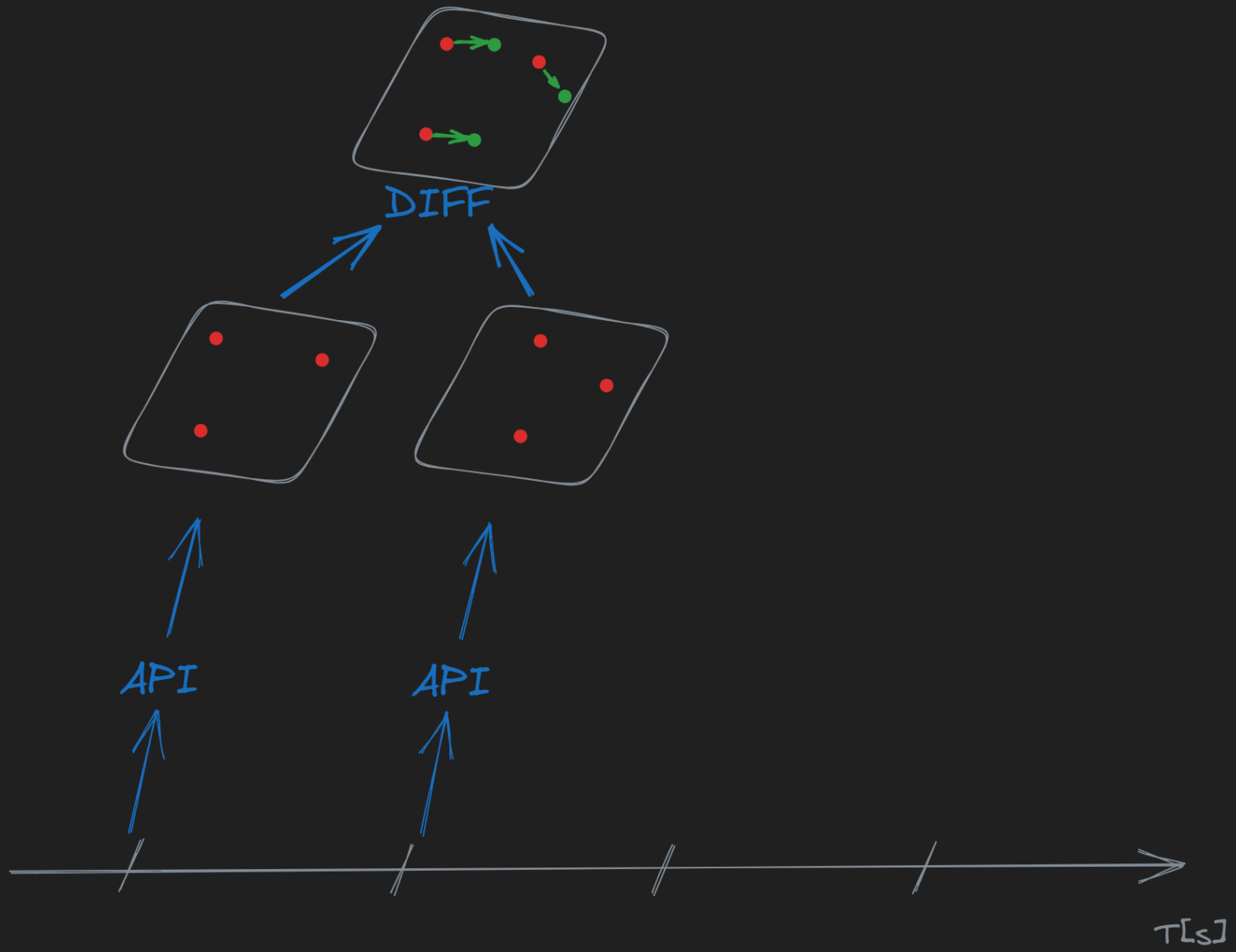
What should our app do

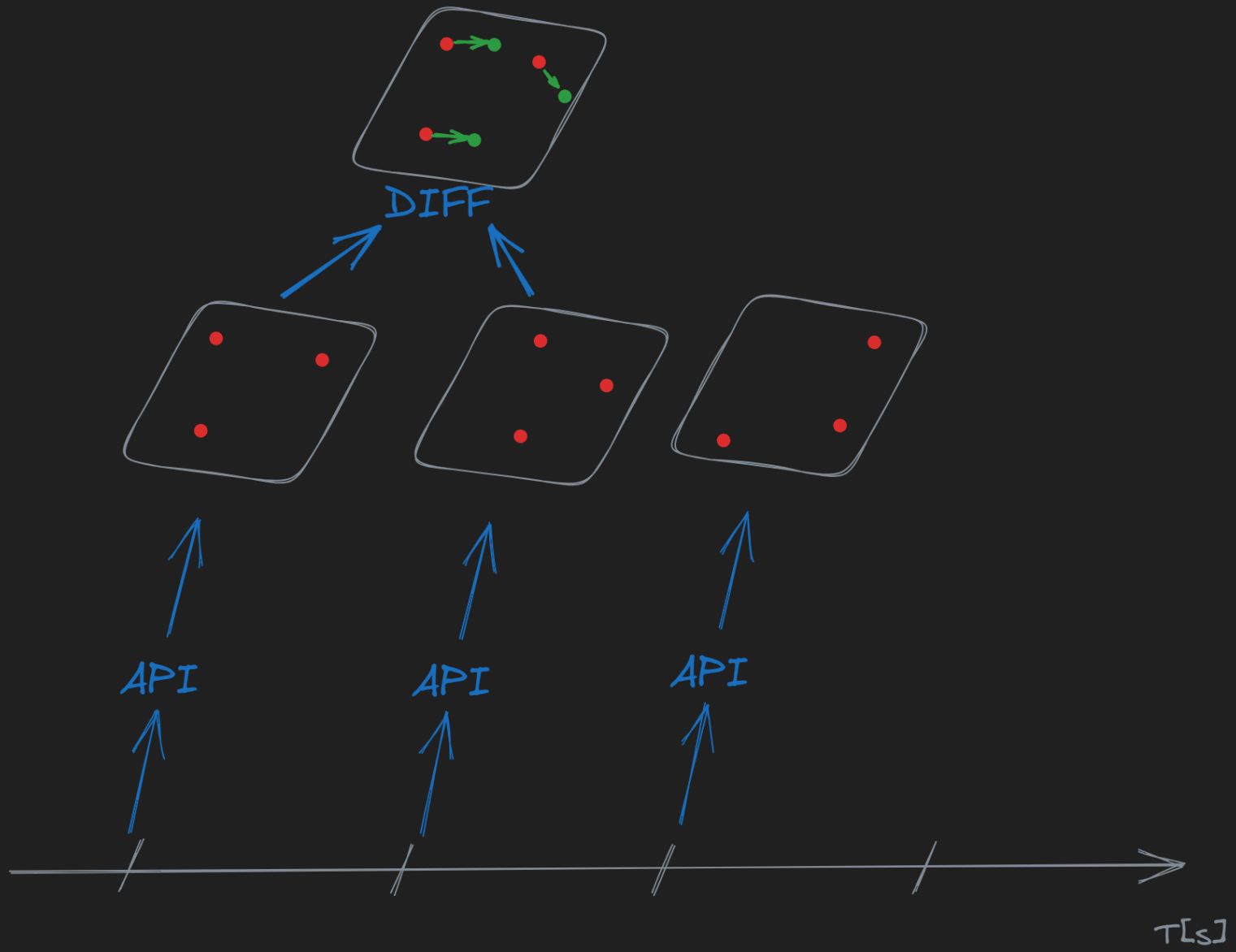
- Upon each `N` seconds call the API for new data
- Compare current result with previous one
  - Find how vehicle moved, calculate diff
- After a fixed amount of updates, show some statistics

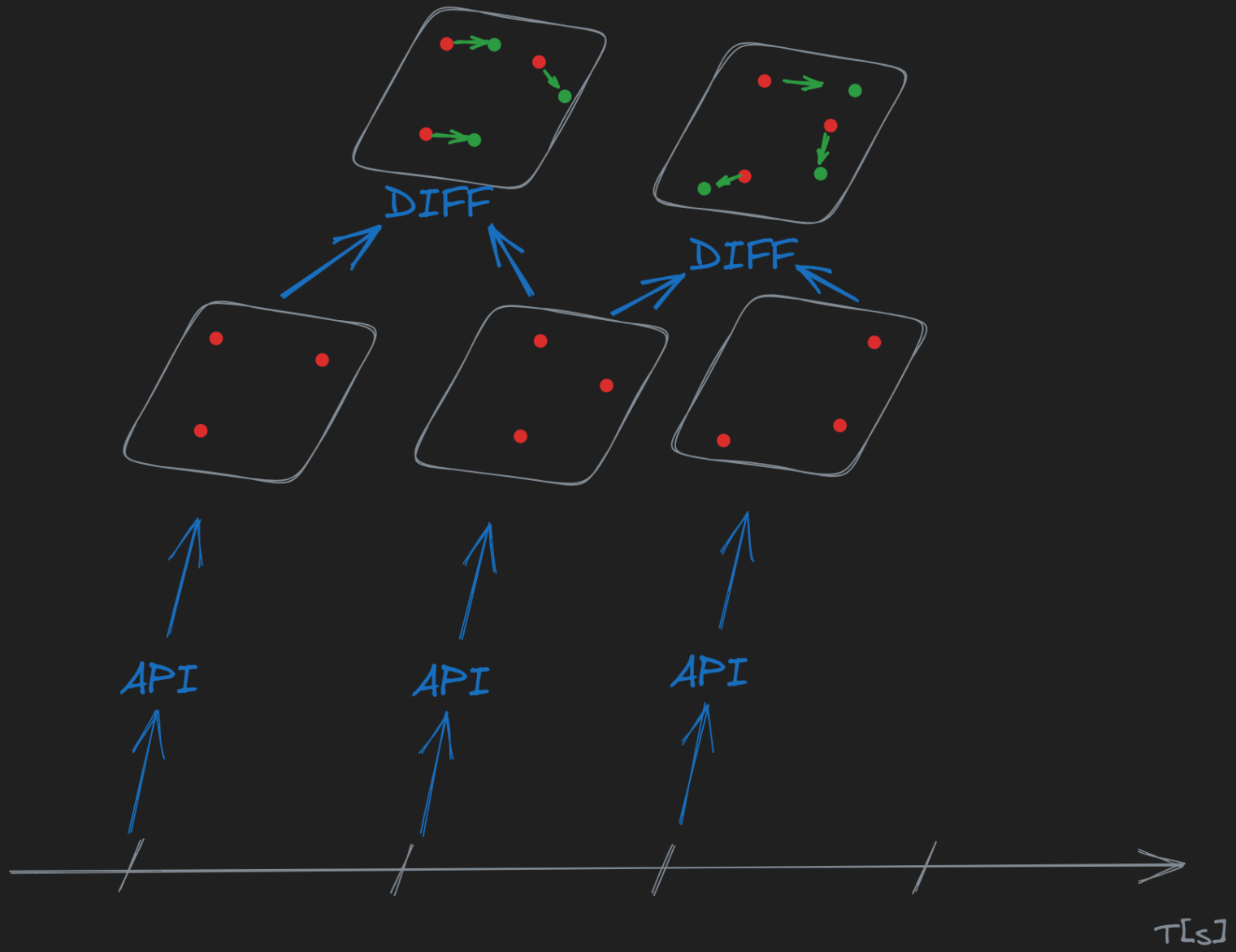
Let's visualize it!











How to implement it

We want to build a stream that:

- Is infinite
- Acts on given time interval like every `N` seconds
- Lists vehicles using `Vehicles[IO].list()`
- Joins results by vehicle `id`
- Calculate the distance using `Vehicle.distance` and the elapsed time
- Build a map of `(Vehicle.LineName, Vehicle.Id) → (Distance, AVG Speed)`

Step by step

Infinite stream that lists vehicles every `N` seconds

Step by step

Infinite stream that lists vehicles every `N` seconds

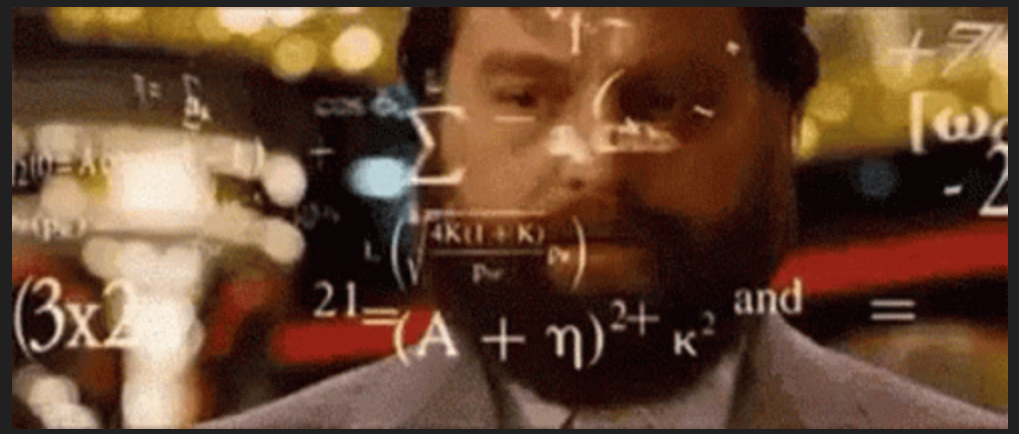
```
1. val interval = 7.seconds
2.
3. def stats(vehicles: Vehicles[IO]): IO[Map[(LineName, Id), VehicleStats]] =
4.   fs2.Stream
5.     .fixedRateStartImmediately[IO](interval)
6.     .evalMap(_ ⇒ vehicles.list())
7.     // TBC
```

Easy right?

Step by step

- Is infinite ✅
- Acts on given time interval like every `N` seconds ✅
- When the time comes - it lists vehicles using `Vehicles[IO].list()` ✅
- Joins previous and current result by vehicle `id`
- Calculate the distance using `Vehicle.distance` and the elapsed time
- Build a map of `(Vehicle.LineName, Vehicle.Id) → VehicleStats`

Step by step



Slide over data, take current and previous measurement and calculate the diff

Step by step

Slide over data, take current and previous measurement and calculate the diff

```
1. def stats(vehicles: Vehicles[IO]): IO[Map[(LineName, Id), VehicleStats]] =
2.   fs2.Stream
3.     .fixedRateStartImmediately[IO](interval)
4.     .evalMap(_ ⇒ vehicles.list())
5.     .sliding(2)
6.     .map(chunk ⇒ calculateDiff(chunk(0), chunk(1))) // That needs explaining
7.     // TBC
```

Step by step

Given two measurements, we want to produce a diff

```
(Vehicle, Vehicle) ⇒ VehiclePositionDiff
```

Step by step

```
1. case class VehiclePositionDiff(
2.   line: Vehicle.LineName,
3.   id: Vehicle.Id,
4.   secondsDuration: Double,
5.   metersDistance: Double
6. )
```

Step by step

```
1. case class VehiclePositionDiff(
2.   line: Vehicle.LineName,
3.   id: Vehicle.Id,
4.   secondsDuration: Double,
5.   metersDistance: Double
6. )
7.
8. object VehiclePositionDiff {
9.   def between(v1: Vehicle, v1: Vehicle): VehiclePositionDiff = {
10.    val duration = secondDuration(v1.measuredAt, v2.measuredAt)
11.    val distance = v1.distance(v2)
12.    VehiclePositionDiff(v1.lineName, v1.id, duration, distance)
13.  }
14. }
```

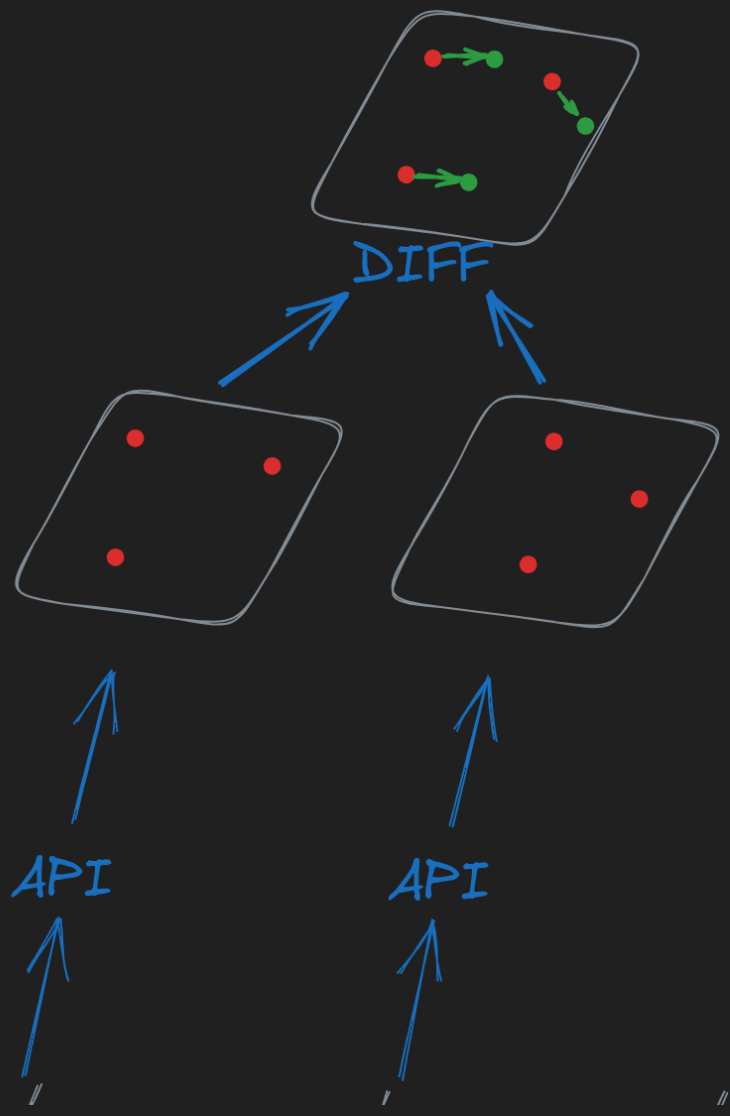
Step by step

Now that we have `(Vehicle, Vehicle) ⇒ VehiclePositionDiff`

Let's do this for multiple measurements

`(Seq[Vehicle], Seq[Vehicle]) ⇒ Seq[VehiclePositionDiff]`

```
(Seq[Vehicle], Seq[Vehicle]) ⇒
Seq[VehiclePositionDiff]
```



Step by step

Important part

```
1. def calculateDiff(snapshot1: Seq[Vehicle], snapshot2: Seq[Vehicle]): Seq[VehiclePositionDiff] =
2.   snapshot1
3.     .join(snapshot2)
4.     .map((v1, v2) ⇒ VehiclePositionDiff.between(v1, v2))
```

Step by step

Important part

```
1. def calculateDiff(snapshot1: Seq[Vehicle], snapshot2: Seq[Vehicle]): Seq[VehiclePositionDiff] =
2.   snapshot1
3.     .join(snapshot2)
4.     .map( (v1, v2) ⇒
5.       VehiclePositionDiff.between(v1, v2)
6.     )
```

Boring stuff, not optimal

```
1. extension (snapshot: Seq[Vehicle]) {
2.   def join(snapshot2: Seq[Vehicle]): Seq[(Vehicle, Vehicle)] =
3.     snapshot.flatMap{ v1 ⇒ snapshot2.collect { case v2 if v2.id == v1.id ⇒ (v1, v2)} }
4. }
```

Step by step

Where are we again?

```
1. def stats(vehicles: Vehicles[IO]): IO[Map[(LineName, Id), VehicleStats]] =
2.   fs2.Stream
3.     .fixedRateStartImmediately[IO](interval)
4.     .evalMap(_ ⇒ vehicles.list())
5.     .sliding(2)
6.     .map(chunk ⇒ calculateDiff(chunk(0), chunk(1))) // basically calculate derivative
7.     // TBC
```

Our stream lists vehicles every `interval` and calculates a list of diffs

Step by step

- Is infinite ✅
- Acts on given time interval like every `N` seconds ✅
- When the time comes - it lists vehicles using `Vehicles[IO].list()` ✅
- Joins previous and current result by vehicle `id` ✅
- Calculate the distance using `Vehicle.distance` and the elapsed time ✅
- Build a map of `(Vehicle.LineName, Vehicle.Id) → VehicleStats`

Step by step

Let's calculate the stats

Easy!

```
1. def stats(vehicles: Vehicles[IO]): IO[Map[(LineName, Id), VehicleStats]] =
2.   fs2.Stream
3.     .fixedRateStartImmediately[IO](interval)
4.     .evalMap(_ ⇒ vehicles.list())
5.     .sliding(2)
6.     .map(chunk ⇒ calculateDiff(chunk(0), chunk(1)))
7.     .take(numberOfSamples)
8.     .fold(Map.empty)(summarize) // 👈 `summarize` needs explaining
9.     .compile
10.    .lastOrError
```

Summarize each step

The `summarize` describes incremental step of building the summary

```scala
def summarize(
  previousSummary: Map[(Vehicle.LineName, Vehicle.Id), VehicleStats],
  nextDiff: Seq[VehiclePositionDiff]
): Map[(LineName, Id), VehicleStats] = {
  val currentSummary =
    nextDiff
      .groupMapReduce
        (d ⇒ (d.line, d.id)) // group by line and id
        (diff ⇒ VehicleStats(diff.metersDistance, diff.secondsDuration)) // map to VehicleStats
        ((a, b) ⇒ a)         // in case there were multiple results, takes first
  Monoid.combine(previousSummary, currentSummary) // Magic 🪄
}
```

Final stream

```
1. def stats(vehicles: Vehicles[IO]): IO[Map[(LineName, Id), VehicleStats]] =
2.   fs2.Stream
3.     .fixedRateStartImmediately[IO](interval)
4.     .evalMap(_ ⇒ vehicles.list())
5.     .sliding(2)
6.     .map(chunk ⇒ calculateDiff(chunk(0), chunk(1)))
7.     .take(numberOfSamples)
8.     .fold(Map.empty)(summarize)
9.     .compile
10.    .lastOrError
```

Let's run it!

```scala
object Main extends IOApp.Simple {
  def run =
    HttpClientFs2Backend
      .resource[IO]()
      .use(backend ⇒ program(backend) *> IO.println("Program finished"))

  private val trams = List("8", "16", "18", "20", "21", "22")
  private val buses = List("124", "145", "149")

  def program(backend: SttpBackend[IO, Any]) = for {
    vehicles  = Vehicles.mpkWrocInstance(backend, buses, trams)
    stats     ← StatsCalculator.stats(vehicles)
    aggregate = StatsCalculator.aggregateLines(stats)
    fastest   = aggregate.maxBy((line, stats) ⇒ stats.avgSpeedKMH)
    slowest   = aggregate.minBy((line, stats) ⇒ stats.avgSpeedKMH)
    avg       = aggregate.values.map(_.avgSpeedKMH).reduce((a, b) ⇒ (a + b) / 2)
    _         ← IO.println(s"Fastest: $fastest")
    _         ← IO.println(s"Slowest: $slowest")
    _         ← IO.println(s"Average: $avg")
  } yield ()

}
```

Results

Captured at 20.03.2024 09:48

Parameters

```
val interval = 9.seconds
val numberOfSamples = 72
```

Stats 📉

- Buses 🚌
  - Fastest: **20.7** km/h - line `149`
  - Slowest: **14.4** km/h - line `124`
- Trams 🚊
  - Fastest: **13.7** km/h - line `21`
  - Slowest: **11.4** km/h - line `16`
-

Stats 📉

- Buses 🚌
  - Fastest: **20.7** km/h - line `149`
  - Slowest: **14.4** km/h - line `124`
- Trams 🚊
  - Fastest: **13.7** km/h - line `21`
  - Slowest: **11.4** km/h - line `16`
- Average: **15.8** km/h

Therefore...

How slow is your tram? 

How slow is your tram? 🚊

Still slower than bike 🚴

But we have learned something! 📖

- Data is not easy to find
- Separation of concerns is important
- Streams can be a nice way to model business logic
- Aquascape is fun!

Thank you!

Keep in touch! 🤝

Blog: blog.michal.pawlik.dev

Linkedin: Michał Pawlik

Github: majk-p

Mastodon: @majkp@hostux.social

Bonus: Monoid magic

Thanks to how monoids compose, you can combine two maps for free

```
//> using toolkit typelevel:latest
import cats.Monoid
val a = Map[String, Int]("foo" → 1, "bar" → 5)
val b = Map[String, Int]("bar" → 3, "baz" → 0)
println(
    Monoid.combine(a, b)
)
```

Bonus: Monoid magic

Thanks to how monoids compose, you can combine two maps for free

```
//> using toolkit typelevel:latest
import cats.Monoid
val a = Map[String, Int]("foo" → 1, "bar" → 5)
val b = Map[String, Int]("bar" → 3, "baz" → 0)
println(
    Monoid.combine(a, b)
)
```

The result is

```
$ scala-cli monoid.sc
Compiling project (Scala 3.3.0, JVM)
Compiled project (Scala 3.3.0, JVM)
Map(bar → 8, baz → 0, foo → 1)
```

Bonus: Monoid magic

So when we provide a `Monoid` for `VehicleStats` like this

```
given Monoid[VehicleStats] with {
  override def combine(x: VehicleStats, y: VehicleStats): VehicleStats =
    VehicleStats(
      x.metersDistance + y.metersDistance,
      x.secondsDuration + y.secondsDuration
    )
  override def empty: VehicleStats = VehicleStats(0,0)
}
```

This works for free

```
Monoid.combine(previousSummary, currentDiffSummary) // combine previous stats with current one together
```