1) HTTP service

GET → www.weather.com / country/US/city/NewYork

← response

URL

params

```
JSON : {
  country : "US"
  city : "NY"
  temp : { [0, 20], [1, 22] .... }
}
```

response

hour → t °C

2) Payment API

POST → api.stripe/v1/make URL

```
{ ..... }
```

params

```
{ status : successful }
```

response

3) add(int x, int y) ⇒ int

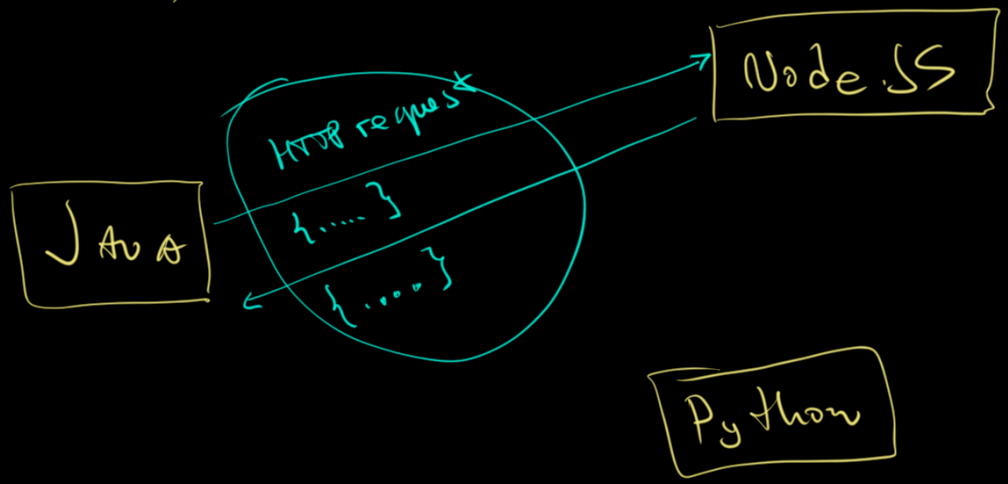
name (url) parameters result

API: 1) address (url, function name, ...)

2) input parameters w/ types) ← encoded somehow (JSON)

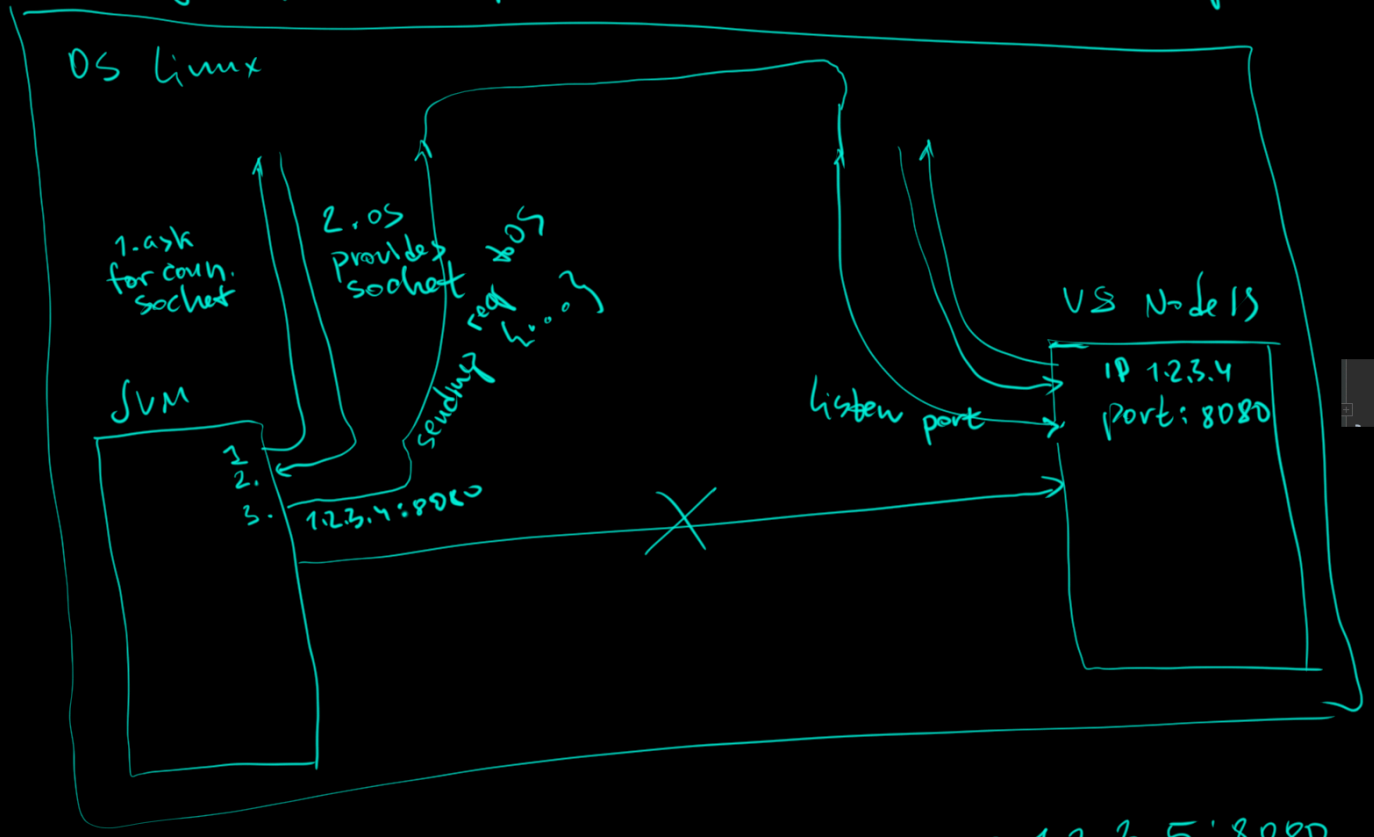
3) output (response) w/ type) →

way to build app



sending request → acquiring every time

listening port → acquire once



port - abstraction of transport

HTTP - ~~encoding~~ protocol

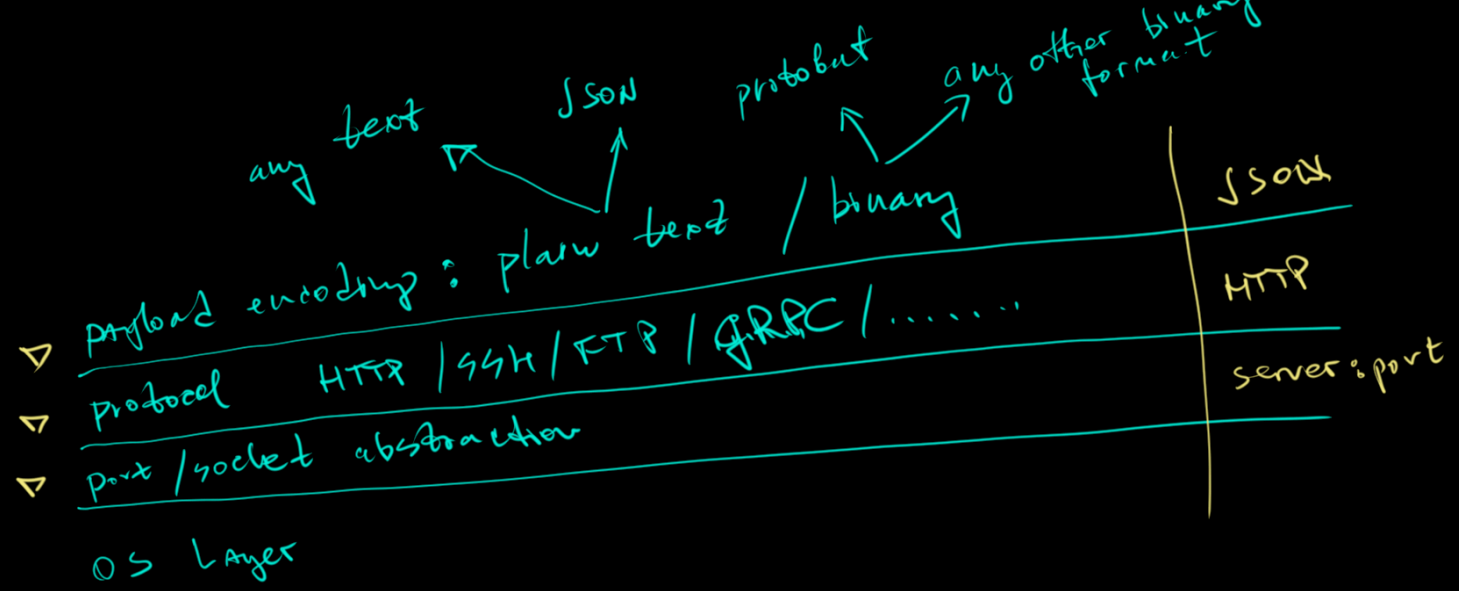
payload - encoding

```
@GetMapping
public Order handle()
```

1.2.3.4 : 53740 → 1.2.3.5 : 8080

1.2.3.10 : 61270 → 1.2.3.5 : 8080

↑ any random number of free (available) ports


any text
JSON
protobuf
any other binary format
payload encoding : plain text / binary
Protocol HTTP / SSH / FTP / gRPC / .....
Port / socket abstraction
OS Layer
JSON
HTTP
server : port

API - way to call any piece of code

- encodings
- protocols
- versioning

```java
// solution2
@GetMapping("/v1/weather")
public Weather provide4(
    @RequestParam("country") String country,
    @RequestParam("city") String city
    ) {
  //......
  return....
}

@GetMapping("/v2/weather")
public Weather provide5(
    @RequestParam("country") String country,
    @RequestParam("city") String city,
    @RequestParam("date") LocalDate date
) {
  //......
  return....
}
```

URL

list of input parameters

output result type

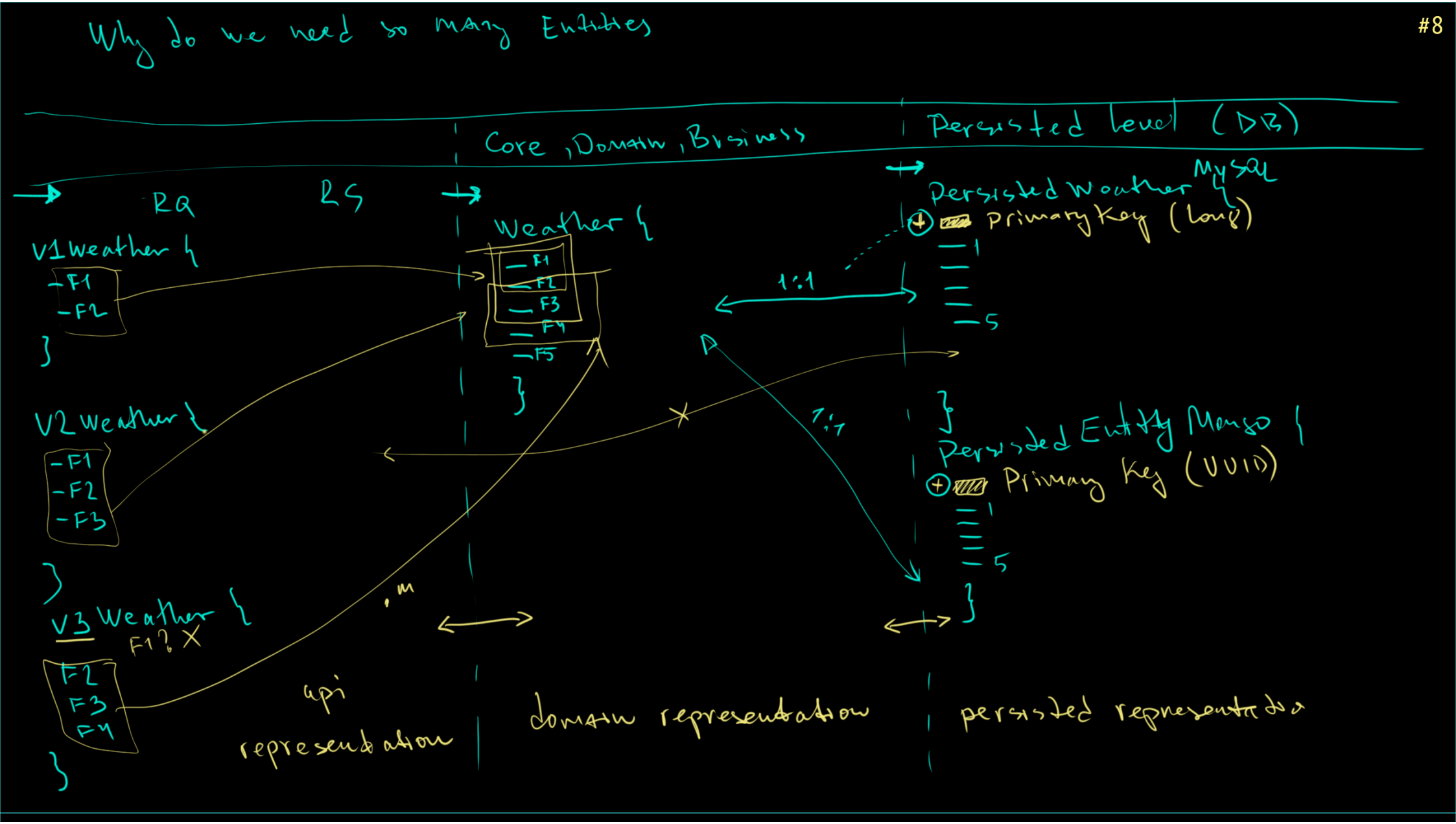
they ~~should~~ must be different

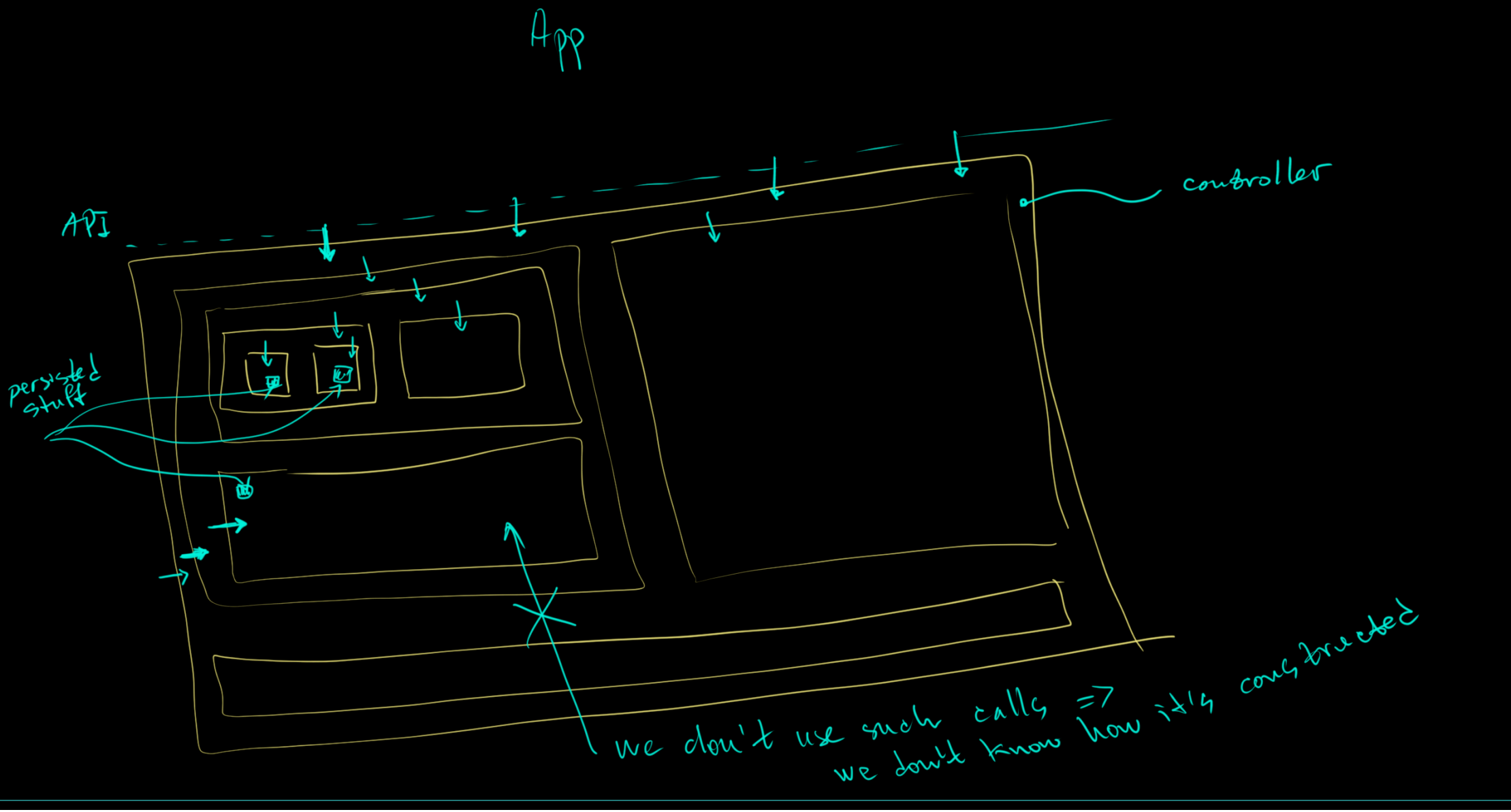
Each API
~~should~~ must have
1) own URL
2) own param list
3) own Result entity

we need at least 3 Entities

1. Weather ito { LocalTime, Temp } - domain description
2. DbWeather / PersistedWeather —"— + key (primary key to distinguish our record w/o ambiguity
3. WeatherV1api
   WeatherV2api
   ⋮

Why do we need so many Entities




App
API
controller
persisted stuff
we don't use such calls =>
we don't know how it's constructed

/v1/controller

/v2/controller

/service

/v1/controller
/v1/service

/v2/controller
/v2/service

/v1/dto
/v2/dto

/v1/dto
/v2/dto

v1 + v2 in the same artifact/repo

V1 repo

V1 artifact

V1 image

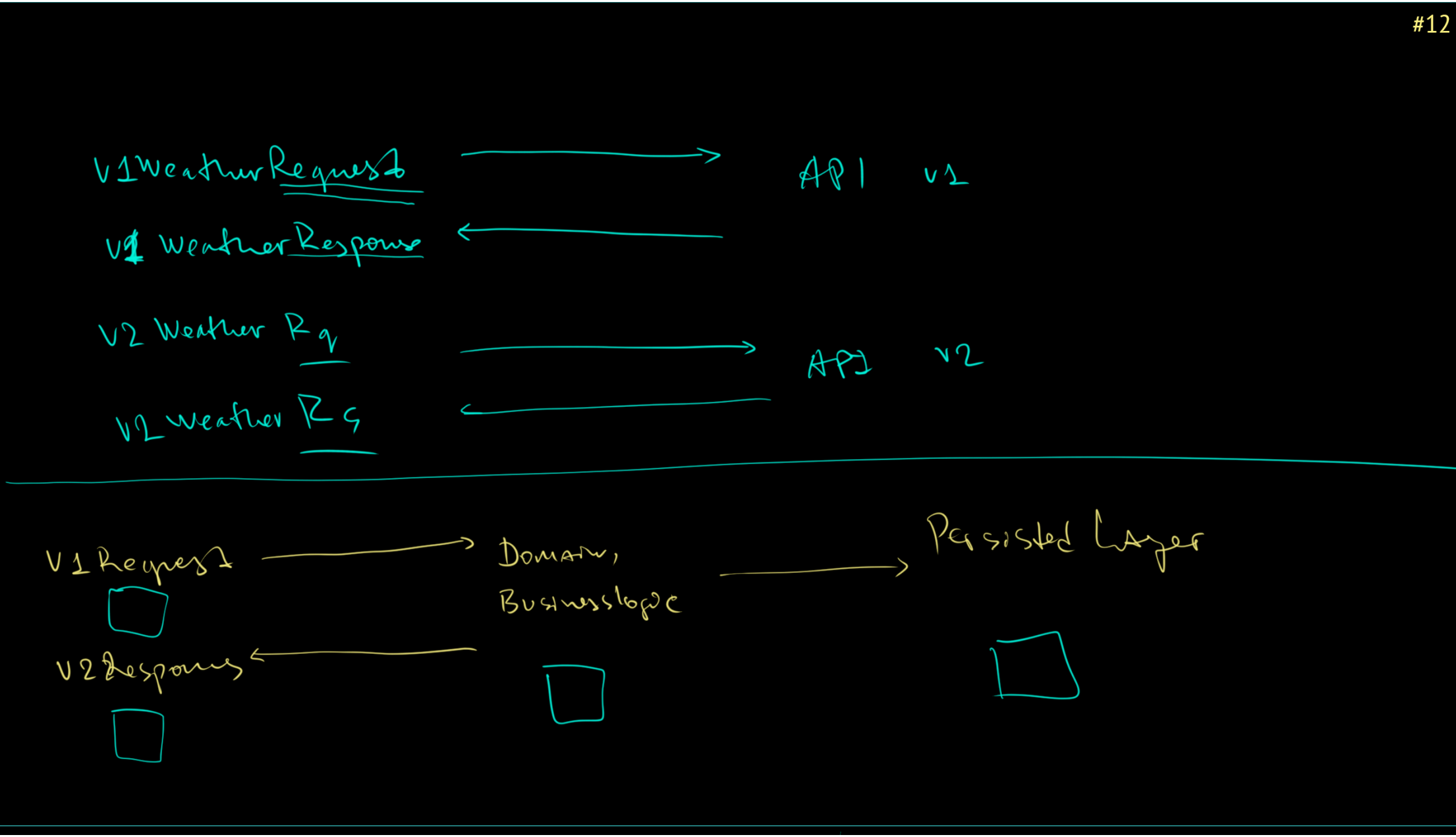
V2 repo

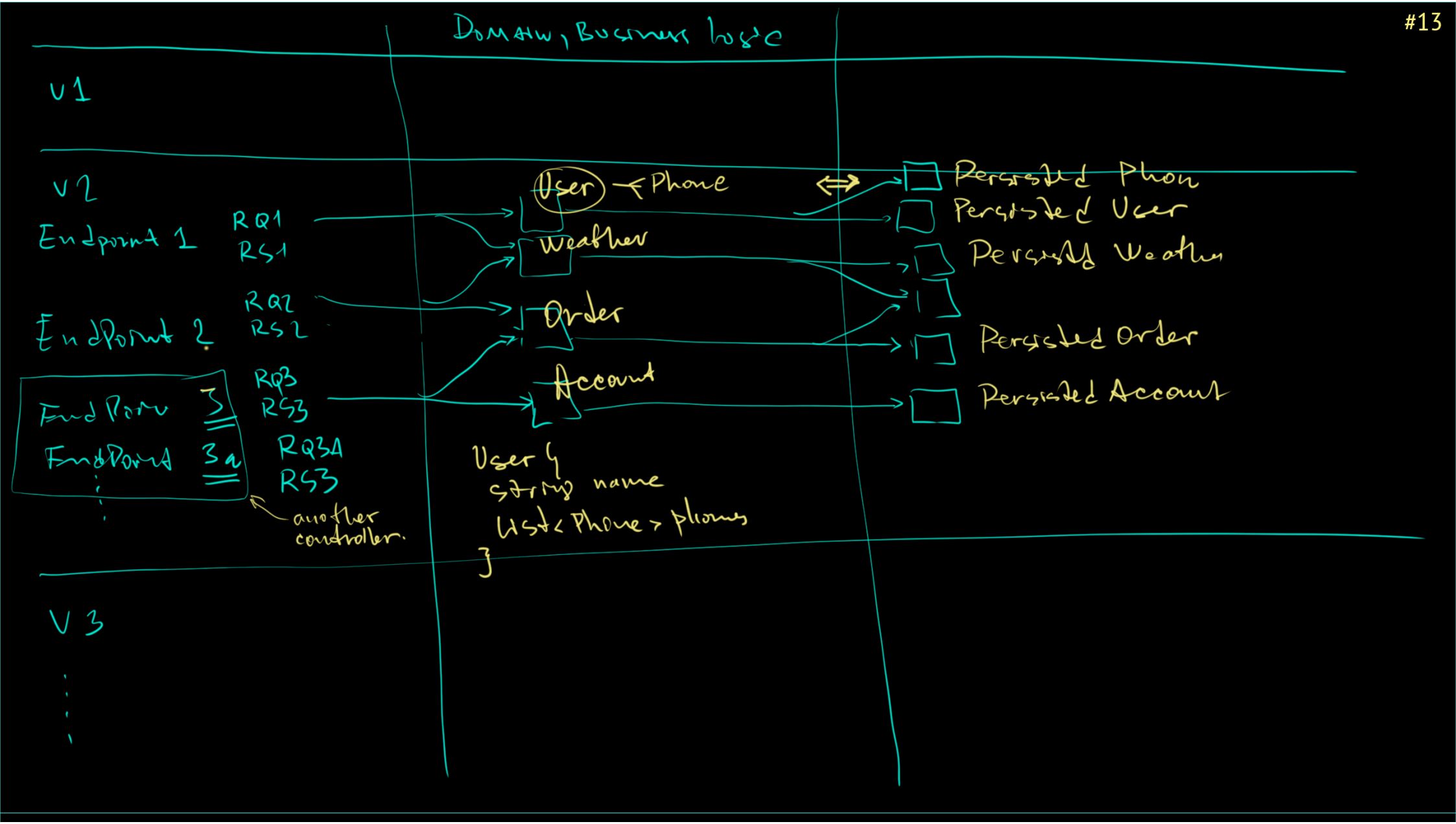
V2 artifact

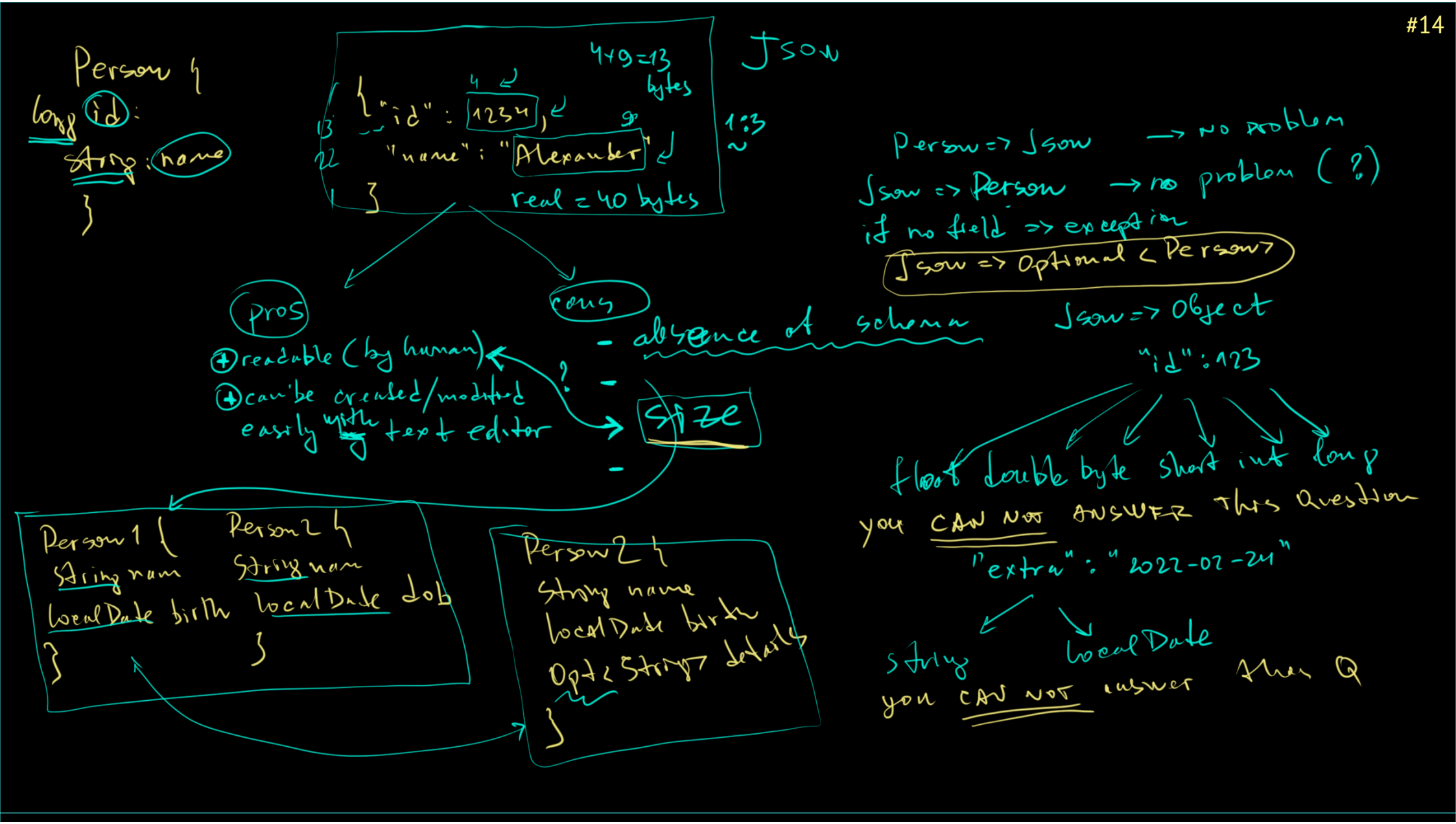
V2 image

=> separate
- balancers
- scaling strategy
- monitoring
- logging
- notifications

- V2 is written with the new framework
  new library
  new team
  new approaches, techniques
  new programming language


V1WeatherRequest
V1 WeatherResponse
API v1
V2 Weather Rq
V2 weather Rs
API v2
V1 Request
Domain, Businesslogic
Persisted Layer
V2 Response


Domain, Business logic
V1
V2
Endpoint 1
RQ1
RS1
EndPoint 2
RQ2
RS2
EndPoint 3
EndPoint 3a
RQ3
RS3
RQ3A
RS3
another controller.
User
Phone
Weather
Order
Account
Persisted Phone
Persisted User
Persisted Weather
Persisted Order
Persisted Account
User {
String name
List<Phone> phones
}
V3


Person {
long id;
String name;
}
{ "id" : 1234,
"name" : "Alexander" }
4+9=13 bytes
real = 40 bytes
Json
1:3
pros
readable (by human)
can be created/modified easily with text editor
cons
absence of schema
?
Size
Person1 {
String nam
LocalDate birth
}
Person2 {
String nam
LocalDate dob
}
Person2 {
String name
LocalDate birth
Opt<String> details
}
Person => Json → no problem
Json => Person → no problem (?)
if no field => exception
Json => Optional<Person>
Json => Object
"id":123
float double byte short int long
you CAN NOT ANSWER this question
"extra" : "2022-02-24"
String
LocalDate
you CAN NOT answer this Q

Controller WeatherService

mapping HttpRequest => (p1, p2, ... pn) -> service

/weather

server location

Client Part

```
class WeatherClient(ipAddr)
    getWeather(String county, String city)
    r = HttpRequest(ipAddr + "/weather" +
                    "?county=" + county + "&city=" + city)

    w = deserialize(r)
    return w
}
```

```
c = new WeatherClient('www.weather.com')
w = c.getWeather('US', 'NY')
```

binary formats

- protobuf
- avro

puts only field number
puts everything in binary

schema

```
syntax = "proto2";

package pbx;

message Student {
  required int32 id = 1;
  required string name = 2;
}
```

it works only with HTTP/2

```
Array(8, 1, 18, 3, 74, 105, 109)
```

```
val student: Student = new Student( id = 1,  name = "Jim")
```

```
serialize object to bytes
val bytes: Array[Byte] = student.toByteArray

deserialize bytes to object
val student2: Student = Student.parseFrom(bytes)
```

Request
/v1/users

→ Result:

[{.....},
{.....},
{.....},

Problems

- amount of data
  - page size
  - page number
  - response can take a lot of time

1. we need to separate] internal and external API

internal
- can provide bunch of data
- can take for a long time
  - pageSize 1..1000

external API
- exactly data requested
- fast
- ...
  - page size 1..20.

Data Filtering

ex: we need users name starting from 'A'

→ select * from users where name like 'A%'

→ .filter(u -> u.name.startsWith('A'))

bad

1. select * from users
2. filtering in JS on UI

good

1. select * from users where name like 'A%'

pros

easy to implement on backend

cons

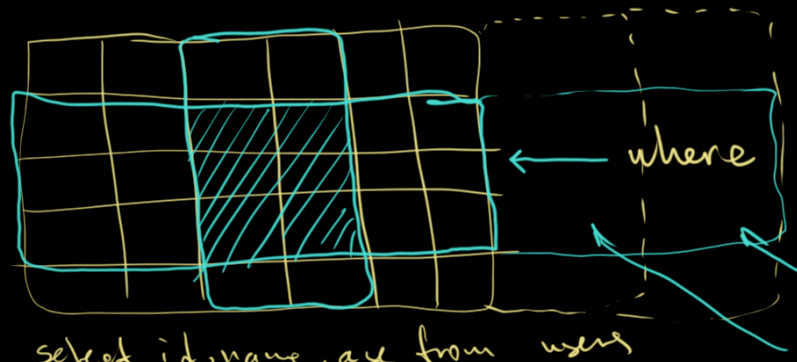
- amount of data is huge
- amount of network traffic is huge
- amount of browser memory allocated is huge

- ~~hard to implement on backend~~
- amount of data is not huge
- we don't blow up the network
- we don't waste the browser memory

- hard to implement on backend

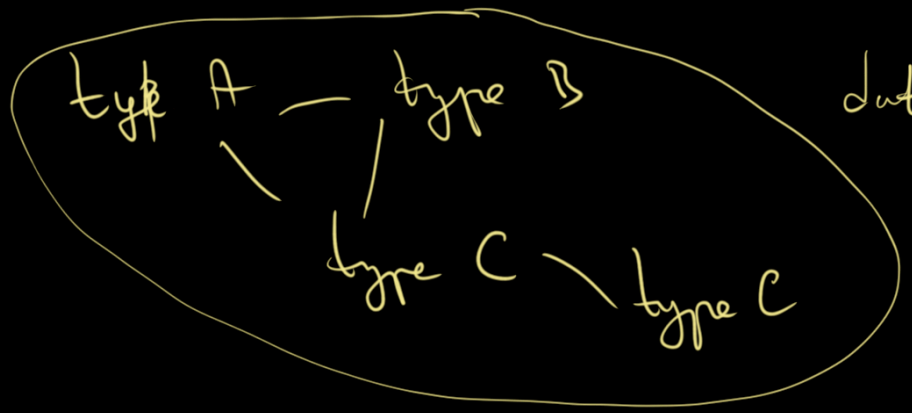
Do we always need it?

⇒ be careful with this list



- be careful about JOINS
  - ↳ every different RQ w diff JOIN must be different endpoint

GraphQL ≈ SQL



class from Java
class name

Describe your data

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

extra: Info

class name

Sequence, Array

Ask for what you want

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

where .....

list of fields to fetch
select *
⇓
select tagline

Get predictable results JSON

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

to make it working

- domain

schema

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

① ⟺ SQL/noSQL any data layer

② domain → SQL

③ result → domain ("Java" classes)?

request ④

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

→ request validation
(domain, request) => Either [Error, Request]

mapping GQL request to "SQL"

running SQL

mapping results → Json taking domain schema

Json as a result

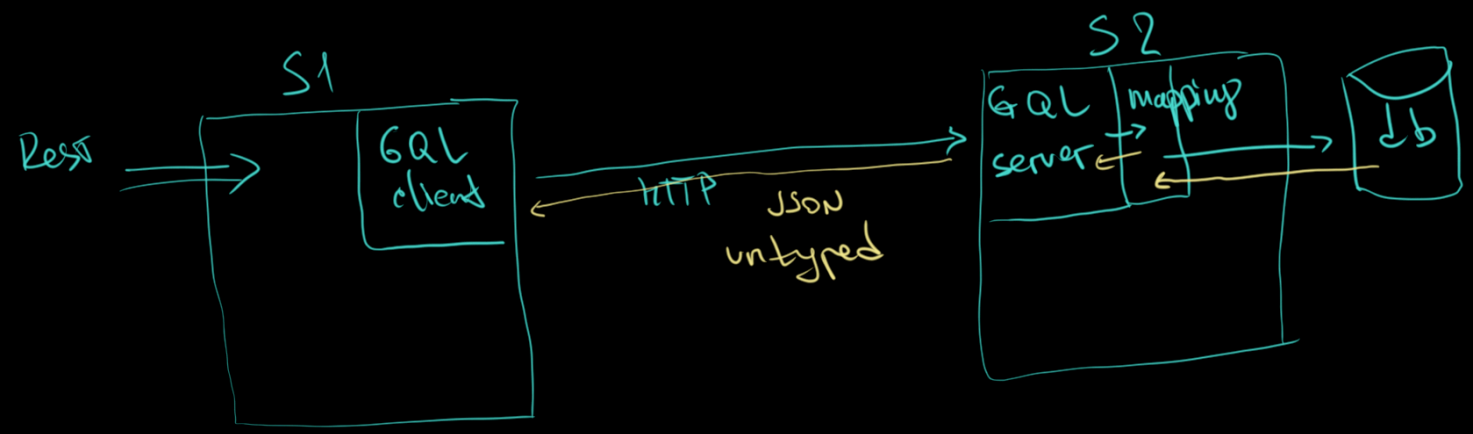
GraphQL is still HTTP endpoint
GraphQL is about payload (POST body)

REST ⟷ GraphQL Query

POST /transfer/a1/a2/am → GraphQL mutation

operation

GraphQL queries – aren't code
– are separate files / strings



```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

→ plain text
it can be shared
between different projects
implemented in different languages
with different libraries

→ often
resides
in separate repo
and maintained
as a subproject

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

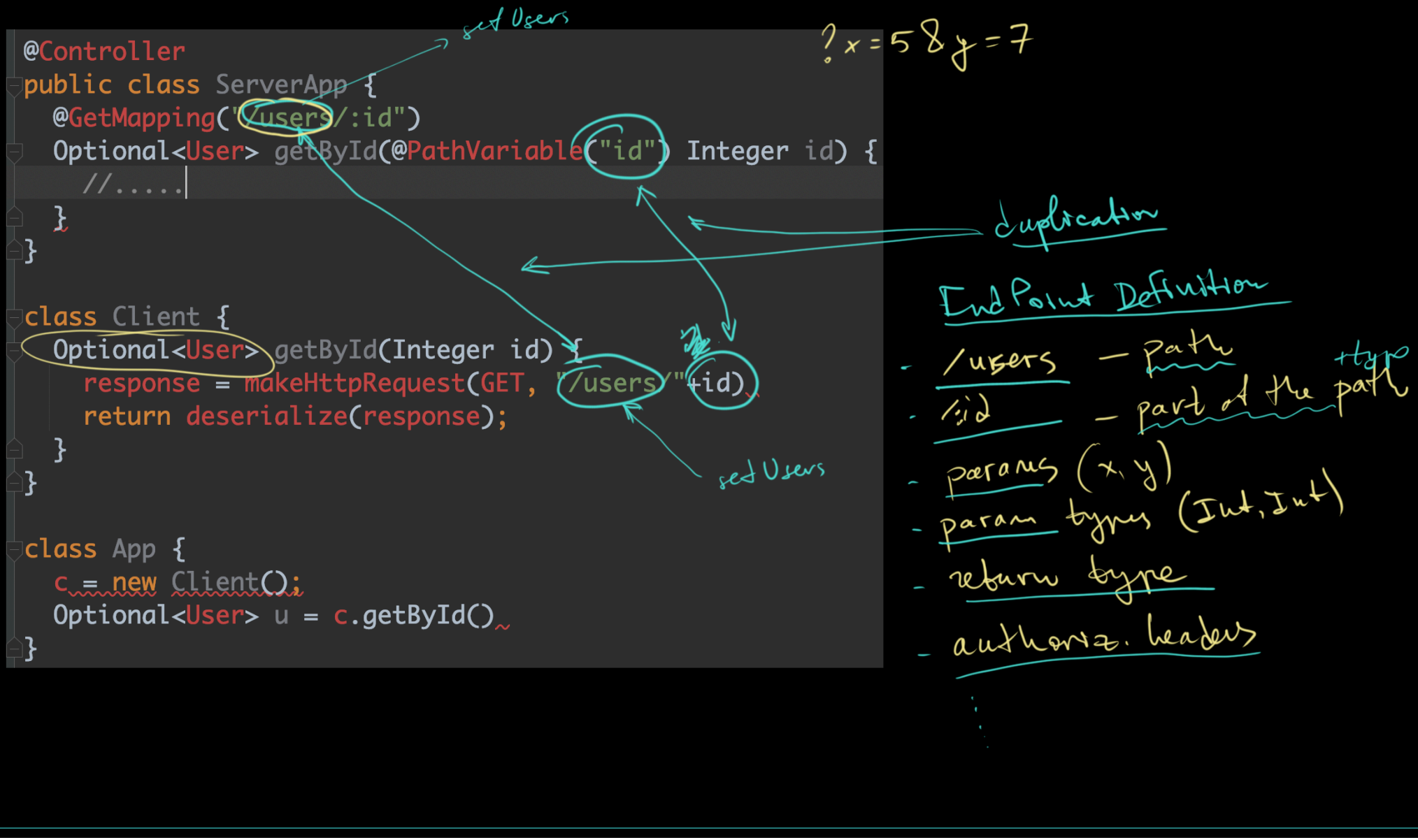
request → plain text

GraphQL rq/rs
CAN'T BE CACHED

Query in
the HTTP body

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

response → plain text


set Users
?x=58y=7
@Controller
public class ServerApp {
@GetMapping("/users/:id")
Optional<User> getById(@PathVariable("id") Integer id) {
//.....
}
}
class Client {
Optional<User> getById(Integer id) {
response = makeHttpRequest(GET, "/users/"+id)
return deserialize(response);
}
}
class App {
c = new Client();
Optional<User> u = c.getById()
}
duplication
End Point Definition
- /users — Path
- /:id — Part of the path
+type
- params (x, y)
- param types (Int, Int)
- return type
- authoriz. headers
set Users

EndPoint ~ plain Java Object

f: (EndPoint) => HttpListener (Controller)
serviceImpl

g: EndPoint => Client (RequestMaker)

```
@GetMapping("/users/:id")
Optional<User> getById(@PathVariable("id") Integer id)
  //.....
}
```

```
Optional<User> getById(Integer id) {
  response = makeHttpRequest(GET, "/users/"+id)
  return deserialize(response);
}
```

functions f & g can be implemented regardless the EndPoint Content

=> we can get rid of duplication because EndPoints description is only in ONE PLACE (EndPoint Java Object)

separate your data / code dealing w/ data