Introduction à GraphQL

@__xuorig__

Marc-André Giroux

APIs @ GitHub

Retournons dans le passé

Nous sommes en 2012

"Endpoint-Based" HTTP APIs

Simplicité

Basé sur les fondations de HTTP

"Endpoint-Based" HTTP APIs

"Endpoint-Based" HTTP APIs

API Client

GET my-api.com/user/1

```
{
  "name": "Mary",
  "age": 30
}
```

API Server

"Endpoint-Based" HTTP APIs

"Endpoint-Based" HTTP APIs

API Client

```
{
  "name": "Mary",
  "friends": [
    "/users/2",
    "/users/3",
    "/users/4",
    "/users/5",
    "/users/6",
  ]
}
```

API Server

"Endpoint-Based" HTTP APIs

API Client

GET my-api.com/user/2

GET my-api.com/user/3

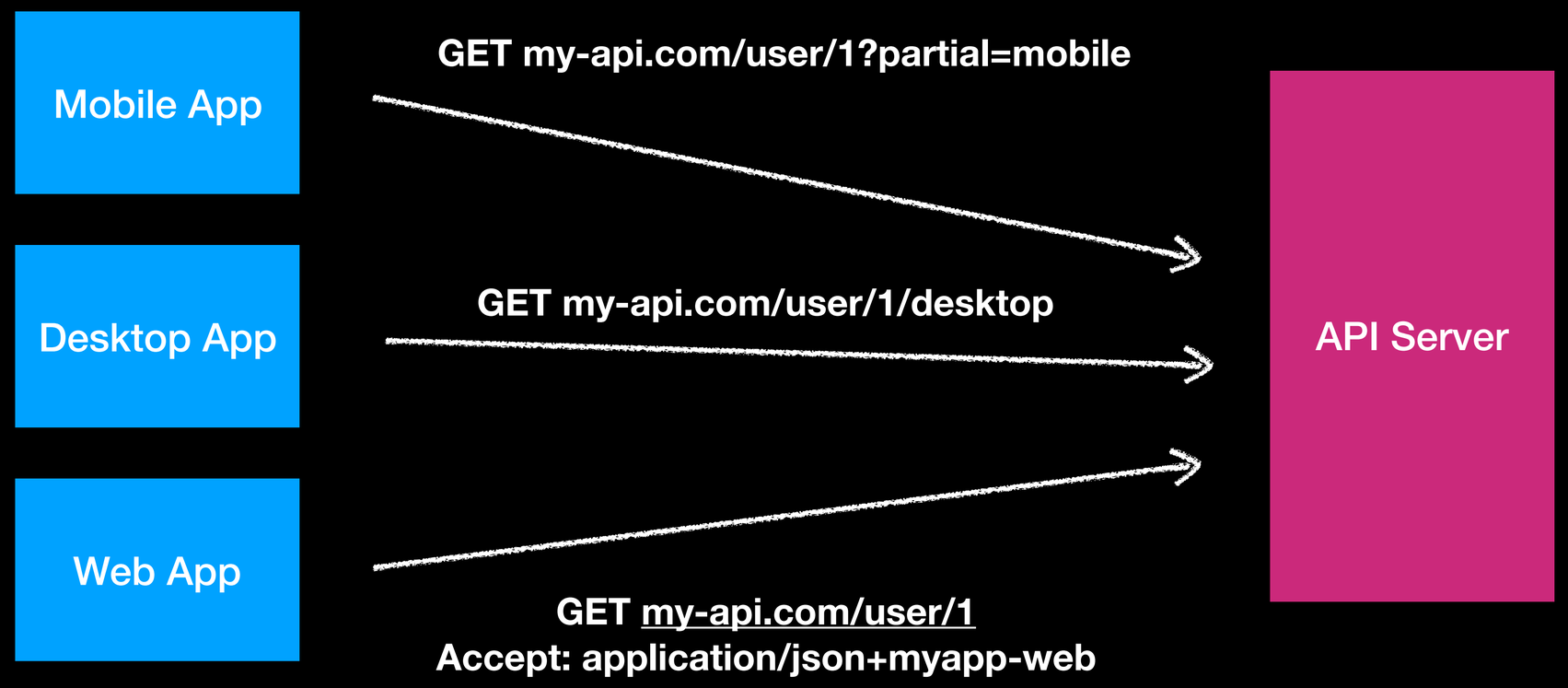
GET my-api.com/user/4

GET my-api.com/user/5

GET my-api.com/user/6

API Server

One-Size-Fits-All APIS (OSFA)



API Client

GET my-api.com/user/1?fields=name,age

```
{
  "name": "Mary",
  "age": 30
}
```

API Server

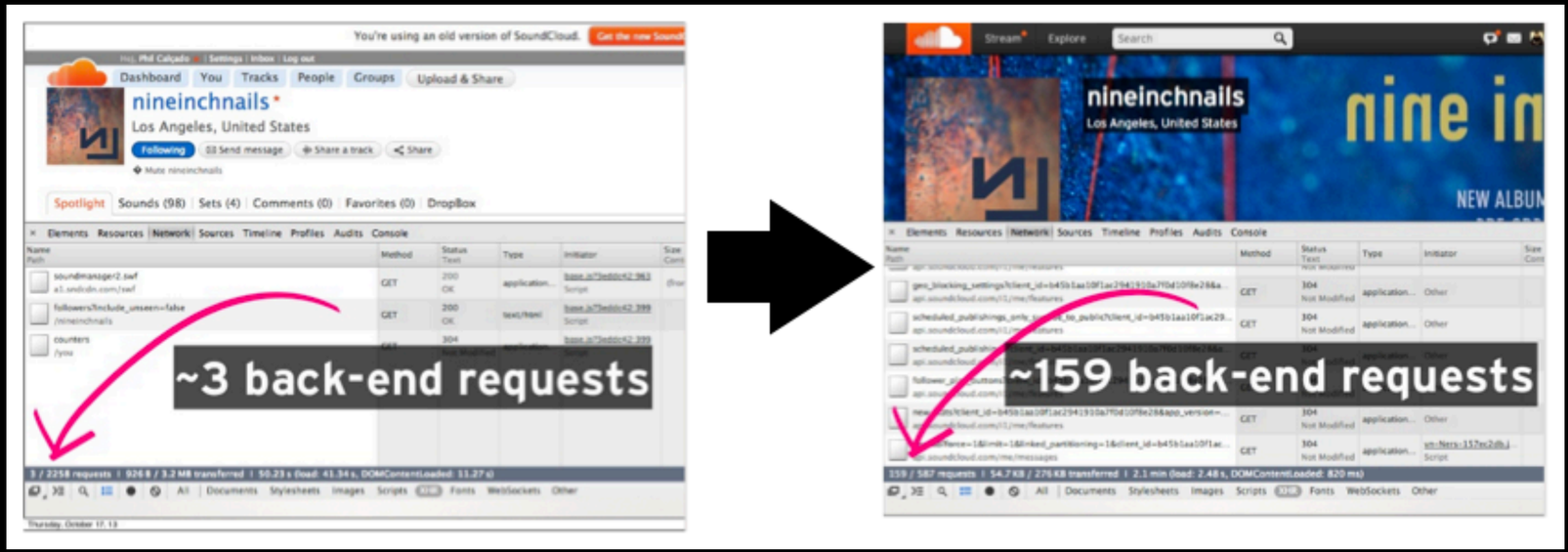
GET /articles?include=author&fields[articles]=title,body&fields[people]=name

APIs "One-Size-Fits-All"

Difficile a entretenir côté serveur

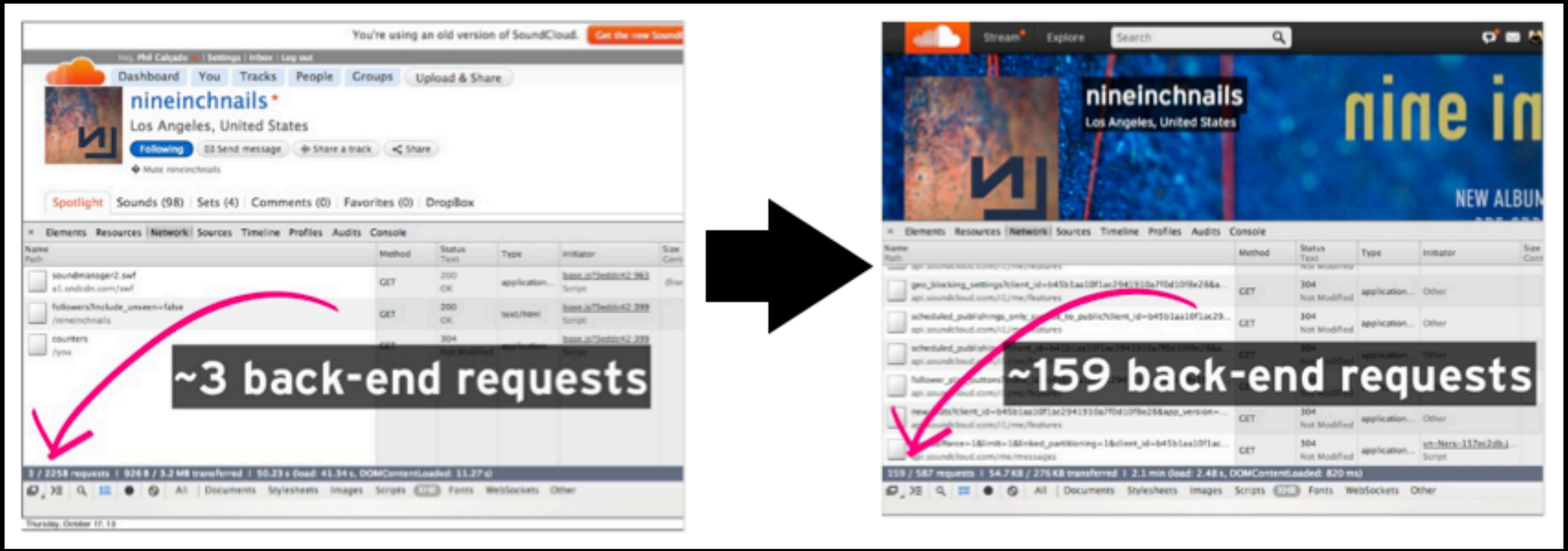
Relation entre serveurs est clients est complexe

Étude de Cas: SoundCloud

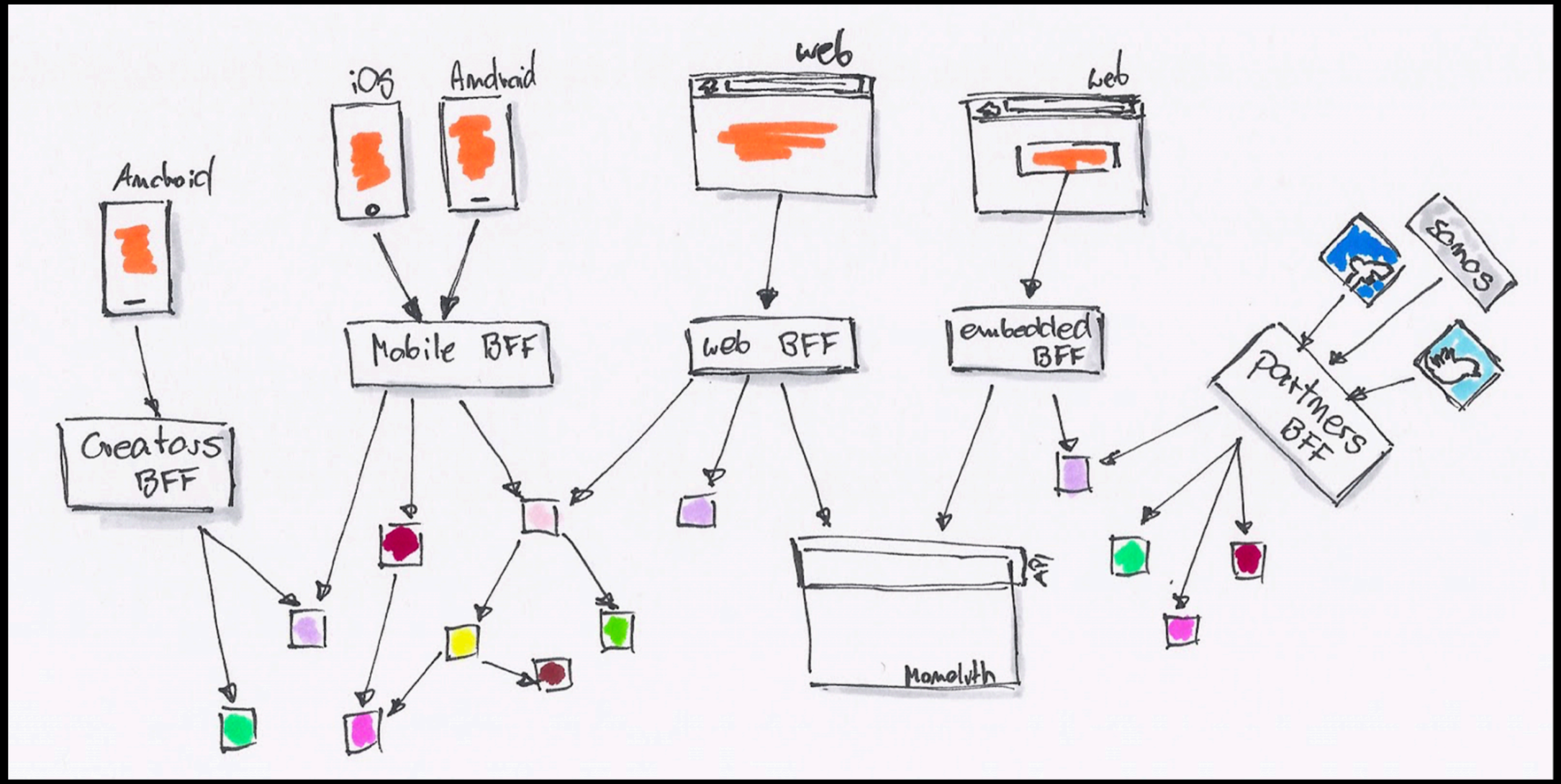


https://philcalcado.com/2019/07/12/some_thoughts_graphql_bff.html

Backend For Frontends

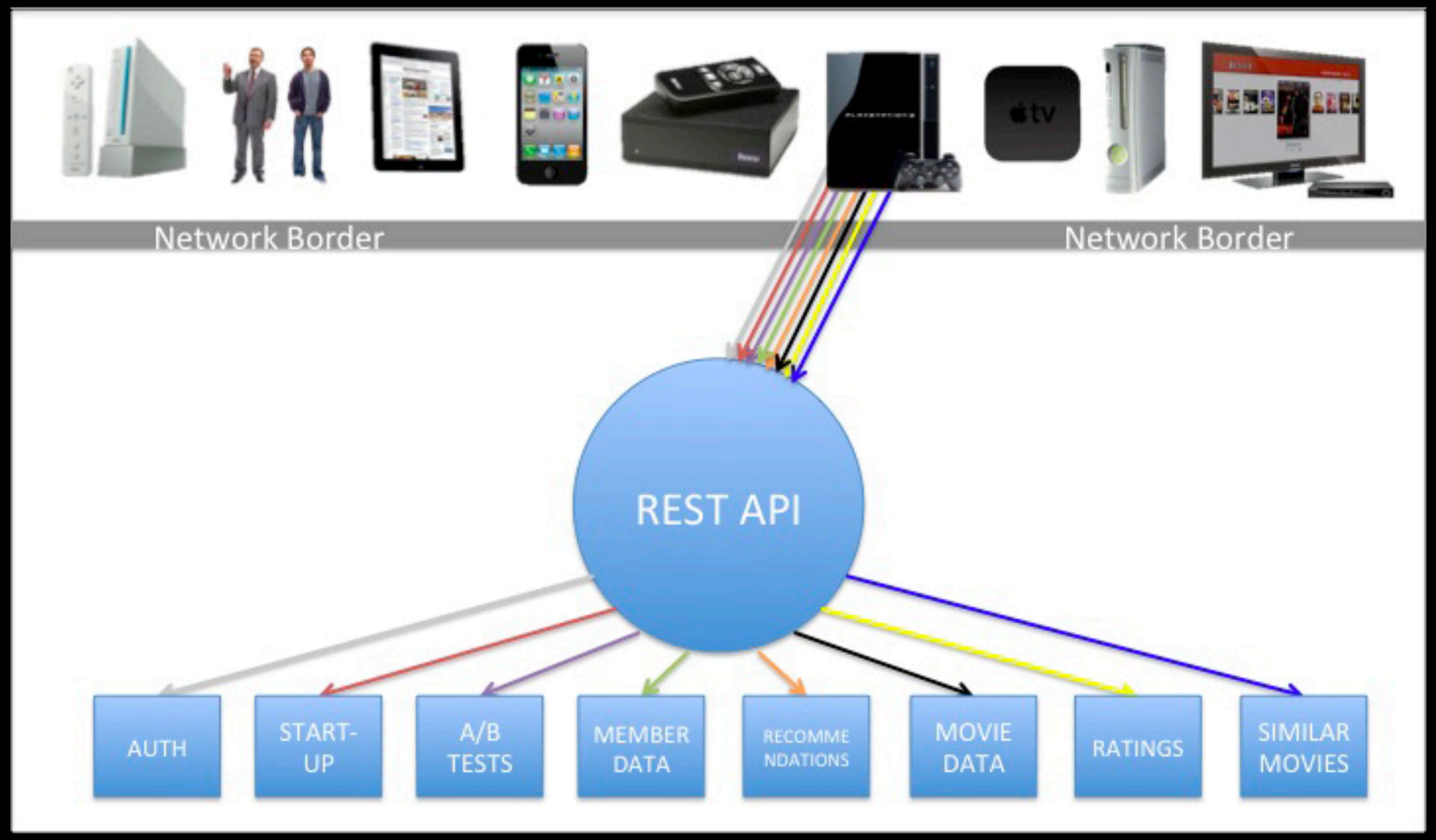


https://philcalcado.com/2019/07/12/some_thoughts_graphql_bff.html

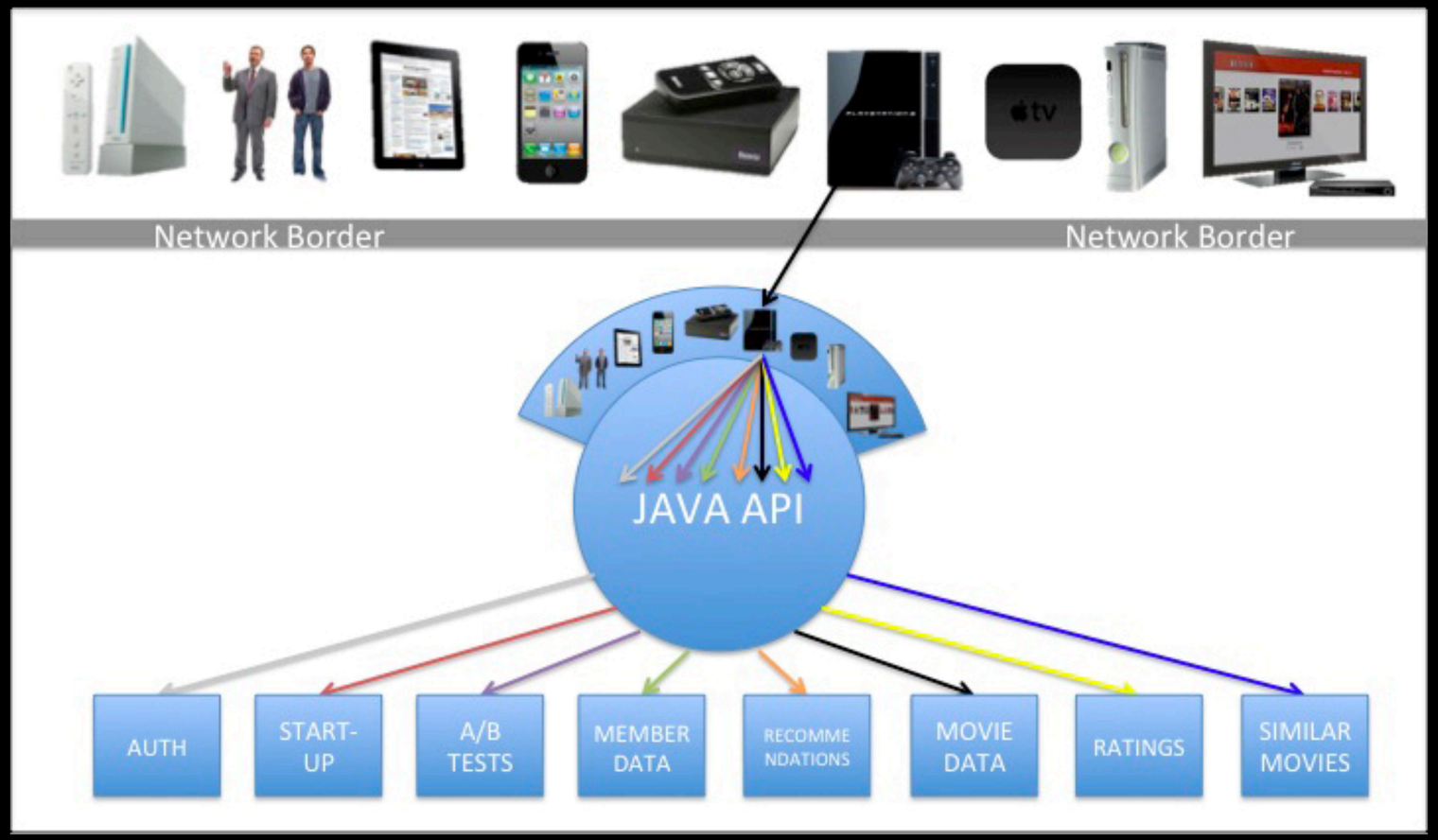


https://philcalcado.com/2019/07/12/some_thoughts_graphql_bff.html

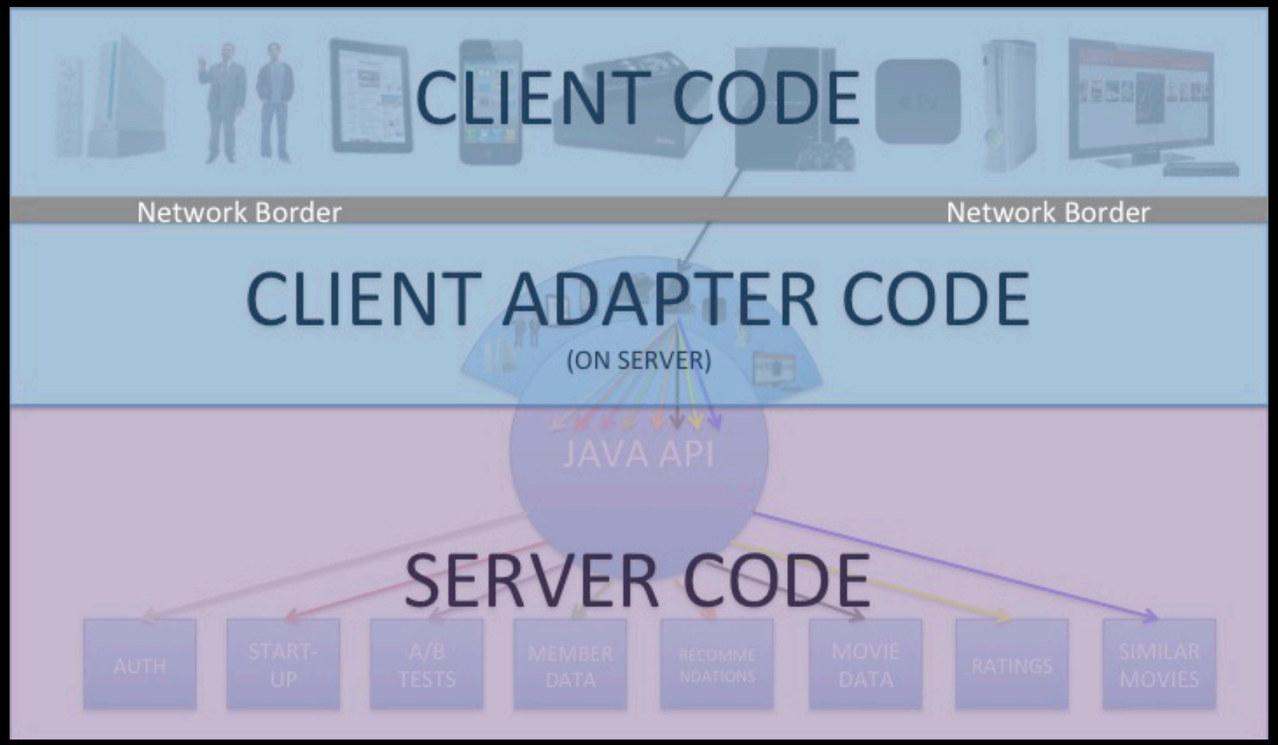
Étude de Cas: Netflix



https://medium.com/netflix-techblog/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d



https://medium.com/netflix-techblog/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d



https://medium.com/netflix-techblog/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d

Quelque chose de commun entre toutes ces approches 🤔

Redéfinir la frontière entre Serveur & Client

Accepter les différences entre Clients

Facebook en 2012

"Back in 2012, we began an effort to rebuild Facebook's native mobile applications"

https://engineering.fb.com/core-data/graphql-a-data-query-language/

"We were frustrated with the differences between the data we wanted to use in our apps and the server queries they required. We don't think of data in terms of resource URLs, secondary keys, or join tables; we think about it in terms of a graph of objects and the models we ultimately use in our apps like NSObjects or JSON."

https://engineering.fb.com/core-data/graphql-a-data-query-language/

GraphQL

Un spécification d'un langage de requête pour APIs.

Une spécification pour un serveur qui détermine comment executer ces requêtes.

```
query {
  me {
    name
  }
}
```

```
{
  "data": {
    "me": {
      "name": "Marc"
    }
  }
}
```

```
query {
  me {
    name
    bestFriends(first: 2) {
      name
    }
  }
}
```

```
{
  "data": {
    "me": {
      "name": "Marc",
      "bestFriends": [{
        "name": "Robert"
      }, {
        "name": "Andrew"
      }]
    }
  }
}
```

Pas de SELECT *

GraphQL Type System

```
type Query {
  me: User
}

type User {
  name: String
  bestFriends(first: Int!): [User]
}
```

```
query {
  me {
    name
    bestFriends(first: 2) {
      name
    }
  }
}
```

```
type Query {
  me: User
}

type User {
  name: String
  bestFriends(first: Int!): [User]
}
```

```
query {
  me {
    name
    bestFriends(first: 2) {
      name
    }
  }
}
```

```
type Query {
  me: User
}

type User {
  name: String
  bestFriends(first: Int!): [User]
}
```

```
query {
  me {
    name
    bestFriends(first: 2) {
      name
    }
  }
}
```

```
type Query {
  me: User
}

type User {
  name: String
  bestFriends(first: Int!): [User]
}
```

```
query {
  me {
    name
    bestFriends(first: 2) {
      name
    }
  }
}
```

```
type Query {
  me: User
}

type User {
  name: String
  bestFriends(first: Int!): [User]
}
```

```
query {
  me {
    name
    bestFriends(first: 2) {
      name
    }
  }
}
```

```
type Query {
  me: User
}

type User {
  name: String
  bestFriends(first: Int!): [User]
}
```

```
query {
  me {
    name
    bestFriends(first: 2) {
      name
    }
  }
}
```

```
type Query {
  me: User
}

type User {
  name: String
  bestFriends(first: Int!): [User]
}
```

Introspection

```
query {
  __schema {
    types {
      name
    }
  }
}
```

```
{
  "data": {
    "__schema": {
      "types": [{
        "name": "Query"
      }, {
        "name": "User"
      }]
    }
  }
}
```

```
{
  "data": {
    "__schema": {
      "types": [{
        "name": "Query"
      }, {
        "name": "User"
      }]
    }
  }
}
```

- Outils
- Documentation
- Validation côté client
- Génération de code

GraphiQL

GraphQL API Explorer | GitHub

developer.github.com/v4/explorer/

Pull Requests | GitHub Team | My PRs | OKTA | GitHubber | Project Boards | Databases | Repos | Teams | Deploy / Builds | API Stuff | ONCALL

GitHub Developer

Docs | Blog | Forum | Versions | Search...

GitHub GraphQL API

Signed in as **xuorig**. You're ready to explore! Sign out

Heads up! GitHub's GraphQL Explorer makes use of your **real, live, production data**.

GraphiQL | Prettify | History

```
query {
  viewer {
    name
  }
}
```

QUERY VARIABLES

```
{}
```

```
{
  "data": {
    "viewer": {
      "name": "Marc-Andre Giroux"
    }
  }
}
```

Documentation Explorer

Search Schema...

A GraphQL schema provides a root type for each kind of operation.

ROOT TYPES

query: Query

mutation: Mutation

launches.js — final

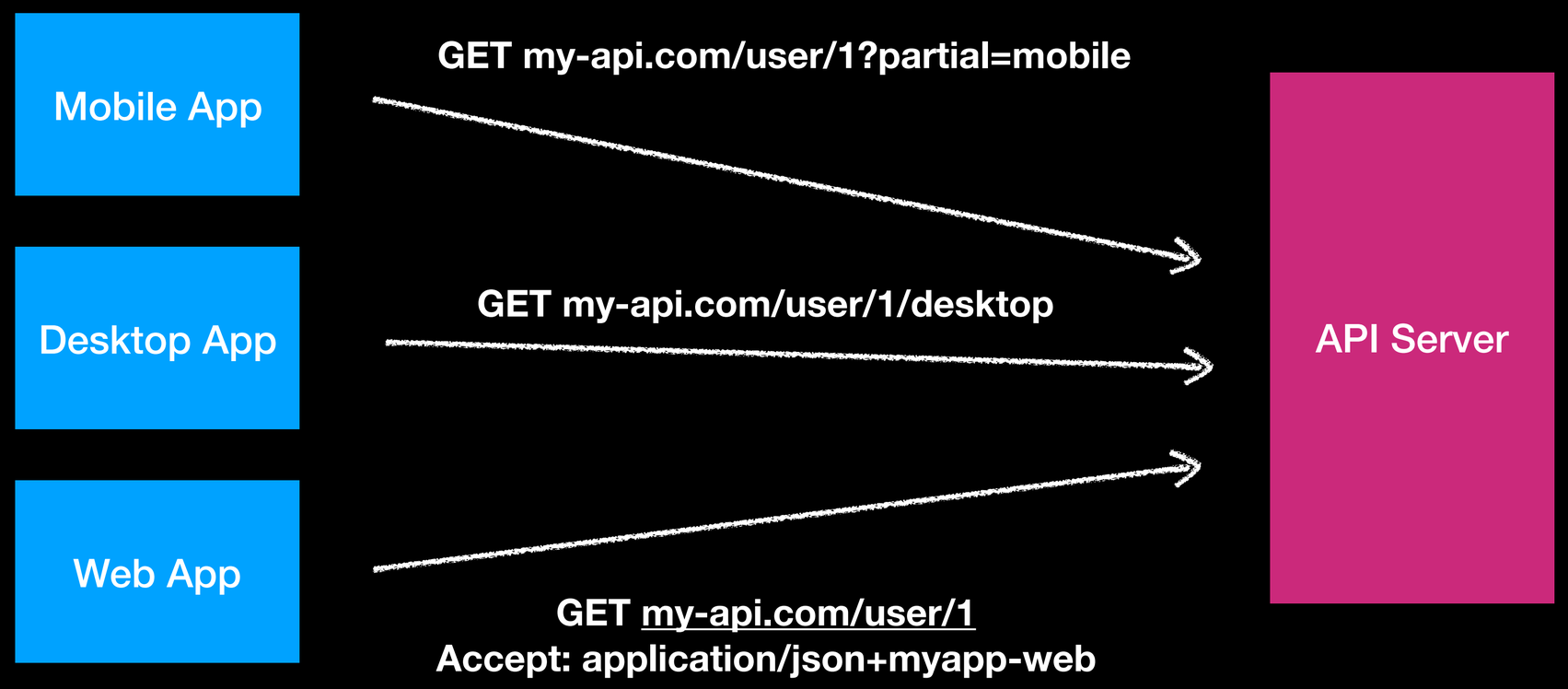
launches.js

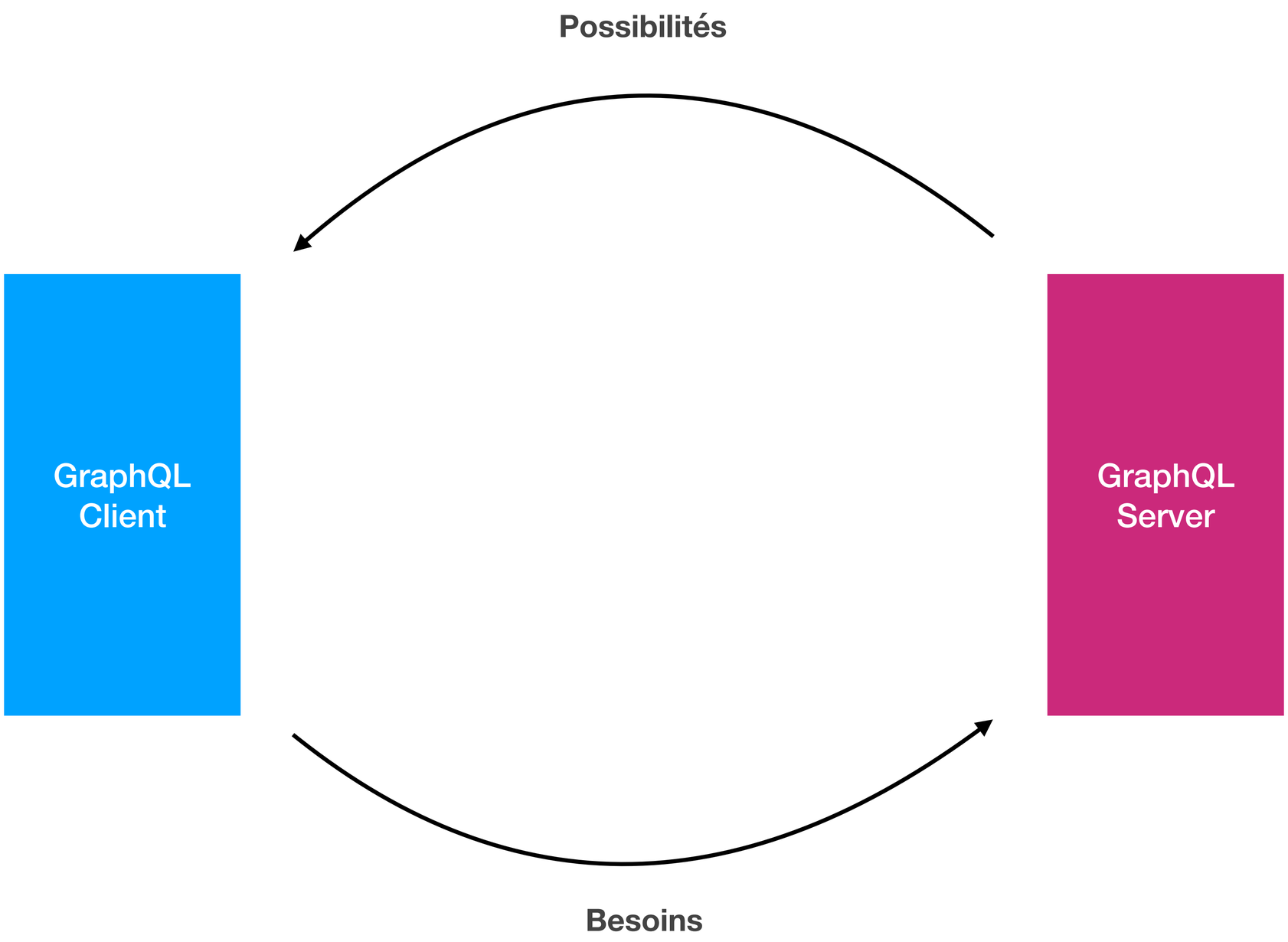
```
6
7   export const LAUNCH_TILE_DATA = gql`
8     fragment LaunchTile on Launch {
9       __typename
10      id
11      isBooked
12
13      mission {
14        name
15        missionPatch
16      }
17    }
18  `;
19
20  export const GET_LAUNCHES = gql`
21    query GetLaunchList($after: String) {
22      launches(after: $after) {
```

master Ln 12, Col 5 Spaces: 2 UTF-8 LF Babel JavaScript ApolloGraphQL Prettier

https://marketplace.visualstudio.com/items?itemName=apollographql.vscode-apollo

Inversion de contrôle


Mobile App
Desktop App
Web App
GET my-api.com/user/1?partial=mobile
GET my-api.com/user/1/desktop
GET my-api.com/user/1
Accept: application/json+myapp-web
API Server


Possibilités
GraphQL
Client
GraphQL
Server
Besoins

```
type Query {
  me: User!
}

type User {
  name: String
  age: Int
  address: Address
  bestFriends(first: Int!): [User]
}

type Address {
  street: String!
  province: Province!
}

enum Province {
  QC
  ON
  BC
}
```

```
type Query {
  me: User!
}

type User {
  name: String
  age: Int
  address: Address
  bestFriends(first: Int!): [User]
}

type Address {
  street: String!
  province: Province!
}

enum Province {
  QC
  ON
  BC
}
```

```
type Query {
  me: User!
}

type User {
  name: String
  age: Int
  address: Address
  bestFriends(first: Int!): [User]
}

type Address {
  street: String!
  province: Province!
}

enum Province {
  QC
  ON
  BC
}
```

```
type Query {
  me: User!
}

type User {
  name: String
  age: Int
  address: Address
  bestFriends(first: Int!): [User]
}

type Address {
  street: String!
  province: Province!
}

enum Province {
  QC
  ON
  BC
}
```

```
type Query {
  me: User!
}

type User {
  name: String
  age: Int
  address: Address
  bestFriends(first: Int!): [User]
}

type Address {
  street: String!
  province: Province!
}

enum Province {
  QC
  ON
  BC
}
```

```
type Query {
  me: User!
}

type User {
  name: String
  age: Int
  address: Address
  bestFriends(first: Int!): [User]
}

type Address {
  street: String!
  province: Province!
}

enum Province {
  QC
  ON
  BC
}
```

```
type Query {
  me: User!
}

type User {
  name: String
  age: Int
  address: Address
  bestFriends(first: Int!): [User]
}

type Address {
  street: String!
  province: Province!
}

enum Province {
  QC
  ON
  BC
}
```

Mutations

Mutations

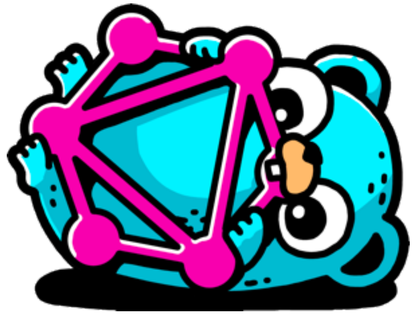
```
type Mutation {
  setMessage(message: String): String
}

type Query {
  getMessage: String
}
```

Mutations

```
mutation {
  createUser(input: { name: "Marc" }) {
    id
    name
  }
}
```

En Pratique

Implementations



GraphQL.js

Example Python: Graphene

```python
from graphene import ObjectType, String, Schema

class Query(ObjectType):
    # this defines a Field `hello` in our Schema with a single Argument `name`
    hello = String(name=String(default_value="stranger"))
    goodbye = String()

    # our Resolver method takes the GraphQL context (root, info) as well as
    # Argument (name) for the Field and returns data for the query Response
    def resolve_hello(root, info, name):
        return f'Hello {name}!'

    def resolve_goodbye(root, info):
        return 'See ya!'

schema = Schema(query=Query)
```

Example Python: Graphene

https://docs.graphene-python.org/

https://github.com/graphql-python/flask-graphql

Exposing the API

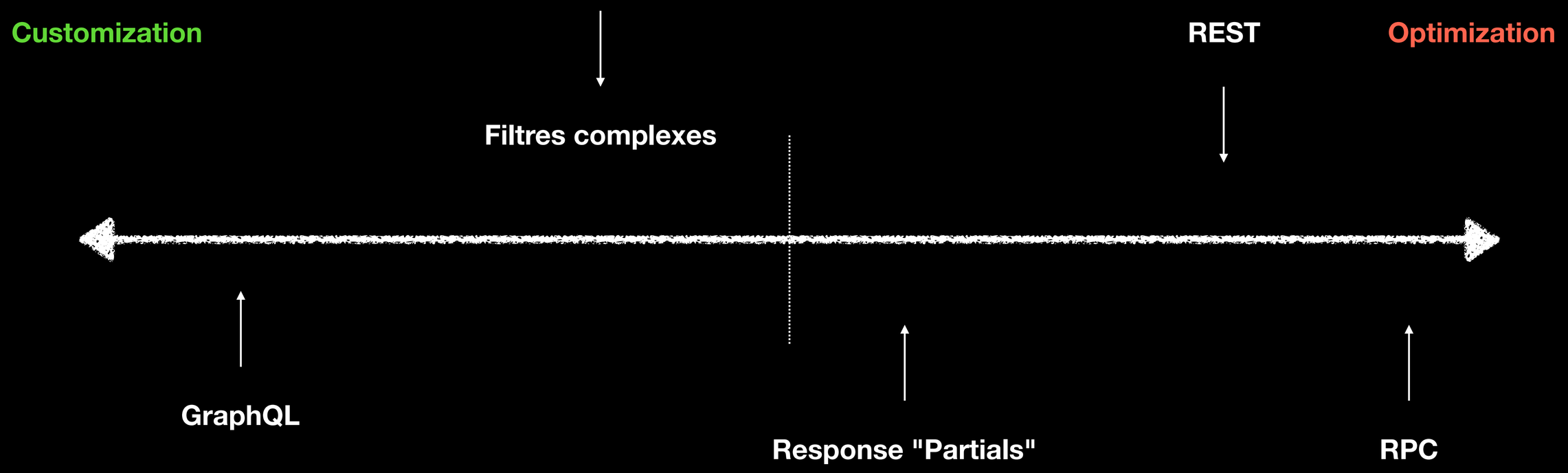
```
POST /graphql

{
  me {
    name
  }
}
```

```
GET /graphql?query=
{
  me {
    name
  }
}
```

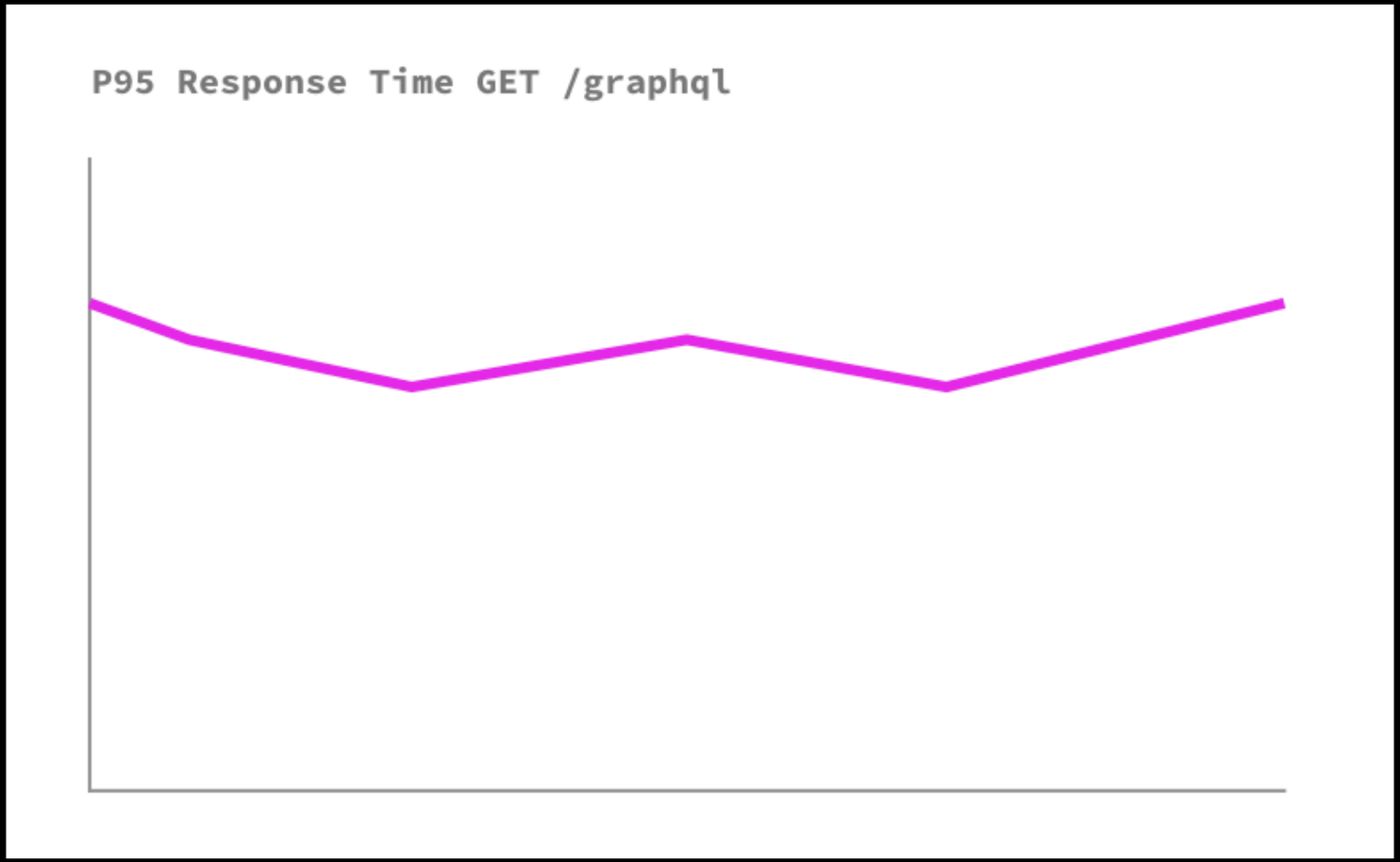
Le côté Obscur

Un grand Pouvoir implique de grandes Responsabilités.

Optimisation vs Flexibilité


Customization
Filtres complexes
REST
Optimization
GraphQL
Response "Partials"
RPC

Performance & Prédictibilité


P95 Response Time GET /graphql

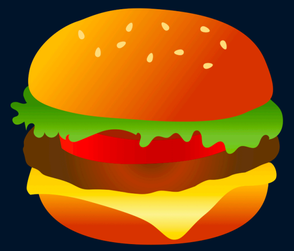
Cache

Documentation (Eh quoi?)

GraphQL & Rest: A burger comparison

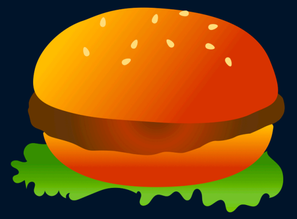
```
https://your-api.com/burger/
```

⇒



```
query getBurger {
  burger {
    bun
    patty
    bun
    lettuce
  }
}
```

⇒



https://drive.google.com/drive/folders/1Lg1T4msi0HL7sWY2XnehKO8vyOyIlg41

Limites & Sécurité

Utilise souvent HTTP comme un "dumb pipe"

200 OK

{ "errors": [...] }

Attention au "hype"

~~GraphQL > REST~~

~~GraphQL c'est le nouveau REST~~

~~GraphQL est plus performant~~

Pour en savoir plus 📖

- Tutoriel: https://www.howtographql.com/
- Blog Posts: https://productionreadygraphql.com
- Twitter: @__xuorig__

Merci!

Questions?
Ping @__xuorig__ on Twitter
ou
mgiroux0@gmail.com