MON1 — Clean API avec GraphQL

10–15 minutes pour comprendre l'essentiel

- Pourquoi GraphQL (vs REST)
- 3–4 features clés avec exemples
- Mini setup Apollo (server + client)
- Bonnes pratiques et limites

Le problème avec REST

Contexte: Applications mobile (réseau social / blog)

```
GET /users/123         → { id, name, email, avatar, bio, ... }
GET /users/123/posts   → [{ id, title, content, date, ... }]
GET /posts/456/stats   → { views, likes, comments }
```

❌ **3 requêtes réseau** • ❌ **Over-fetching** (données inutiles) • ❌ **Under-fetching** (stats manquantes)

La solution GraphQL

Une seule requête, données exactes:

```
query UserProfile($id: ID!) {
  user(id: $id) {
    name
    avatar
    posts(first: 5) {
      title
      date
      stats {
        views
        likes
      }
    }
  }
}
```

1 requête • Données exactes • Typé

GraphQL aujourd'hui

Adoption en production :

- **Netflix** : Fédération de 100+ microservices
- **GitHub, Shopify, Meta** : APIs publiques
- **Apollo Federation** : Un schema unifié sur plusieurs équipes

Écosystème mature :

- Tooling complet (IDE, tests, monitoring)
- Caches intelligents (Apollo, Relay)
- Types TypeScript générés depuis le schéma

➡️ **Plus qu'un simple remplaçant des REST APIs**

Vocabulaire minimal

- Opérations: Query (lecture), Mutation (écriture)
- Schéma: types, interfaces, fragments
- Résolveurs: "où et comment" récupérer les données
- Erreurs partielles: `data` + `errors` dans la même réponse

Exemple (GitHub GraphQL API):

```
query {
  viewer {
    login
  }
}
```

Aliases et fragments

Fragments = réutiliser des champs

```
fragment common on Organization {
  login
  description
}

query {
  facebook: organization(login: "facebook") {
    ...common
  }
  microsoft: organization(login: "microsoft") {
    ...common
  }
}
```

Avantages: Une seule requête, réponses structurées

Interfaces et polymorphisme

Même requête, type différent → **inline fragments**

```
query ($login: String!) {
  repositoryOwner(login: $login) {
    __typename
    ... on User {
      company
      bio
    }
    ... on Organization {
      name
      description
    }
  }
}
```

Idée clé: Le client s'adapte au type concret sans changer d'endpoint

Pagination "Connection" (cursor-based)

Stable et performante (GitHub v4)

```
query ($login: String!, $after: String) {
  user(login: $login) {
    repositories(first: 5, after: $after) {
      edges {
        node {
          name
        }
        cursor
      }
      pageInfo {
        hasNextPage
        endCursor
      }
    }
  }
}
```

Mutations (et cache côté client)

```
mutation AddBook($input: AddBookInput!) {
  addBook(input: $input) {
    book {
      id
      title
      author
    }
  }
}
```

Bonnes pratiques:

- Inputs typés + payload clair
- Optimistic UI + mise à jour du cache
- AuthZ/permissions dans les résolveurs

Mini projet Apollo (server + client)

Server (Apollo Server v4)

```
const typeDefs = `#graphql
  type Book { title: String! author: String! }
  type Query { books: [Book!]! }
`;
const books = [
  { title: "Le Seigneur des anneaux", author: "Tolkien" },
  { title: "L'Étranger", author: "Camus" },
];
const resolvers = { Query: { books: () => books } };
```

Client React + Apollo

```
const GET_BOOKS = gql`
  query {
    books {
      title
      author
    }
  }
`;

const { loading, error, data } = useQuery(GET_BOOKS);
```

Ressources

- GraphQL
- Apollo Server
- Apollo Client
- GitHub GraphQL