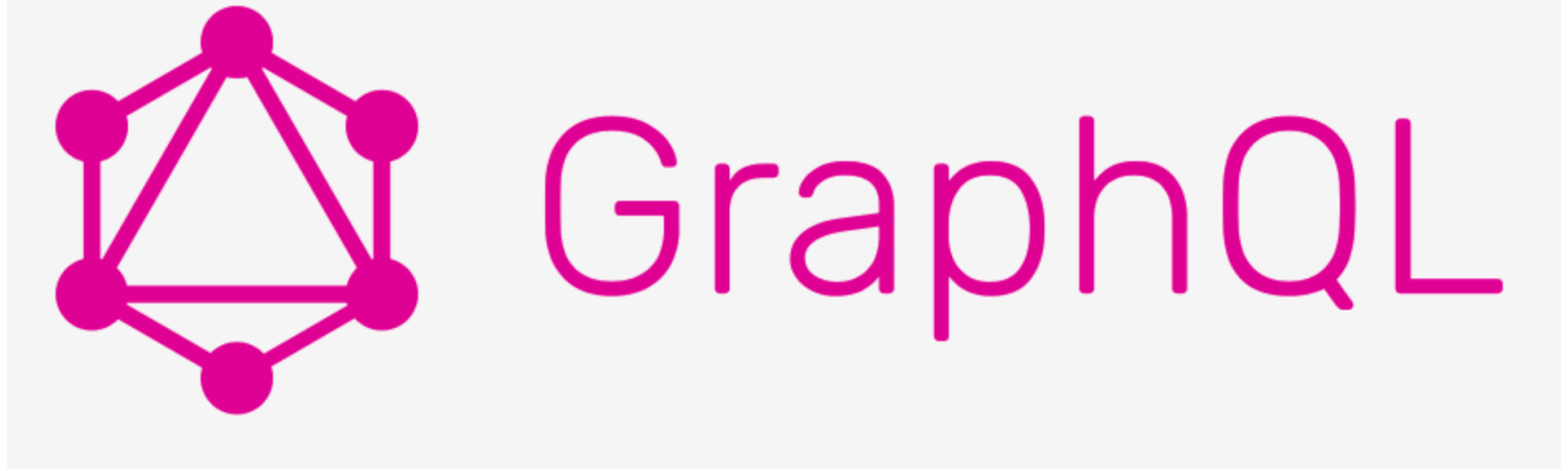
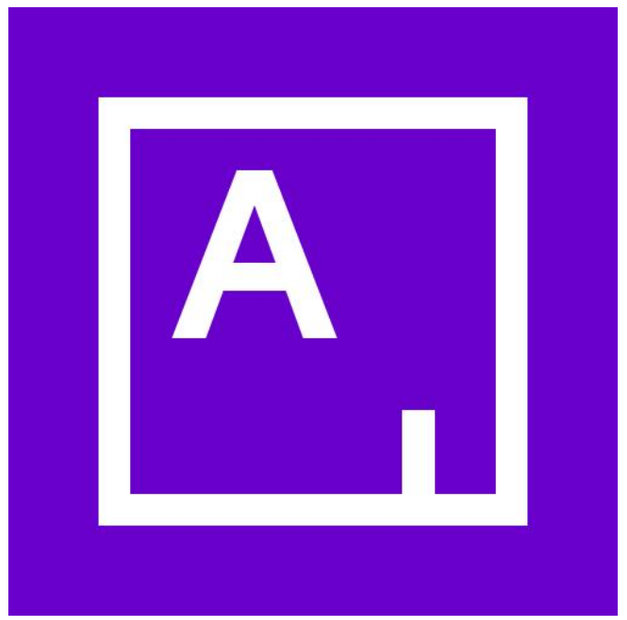


(A very, very brief introduction)

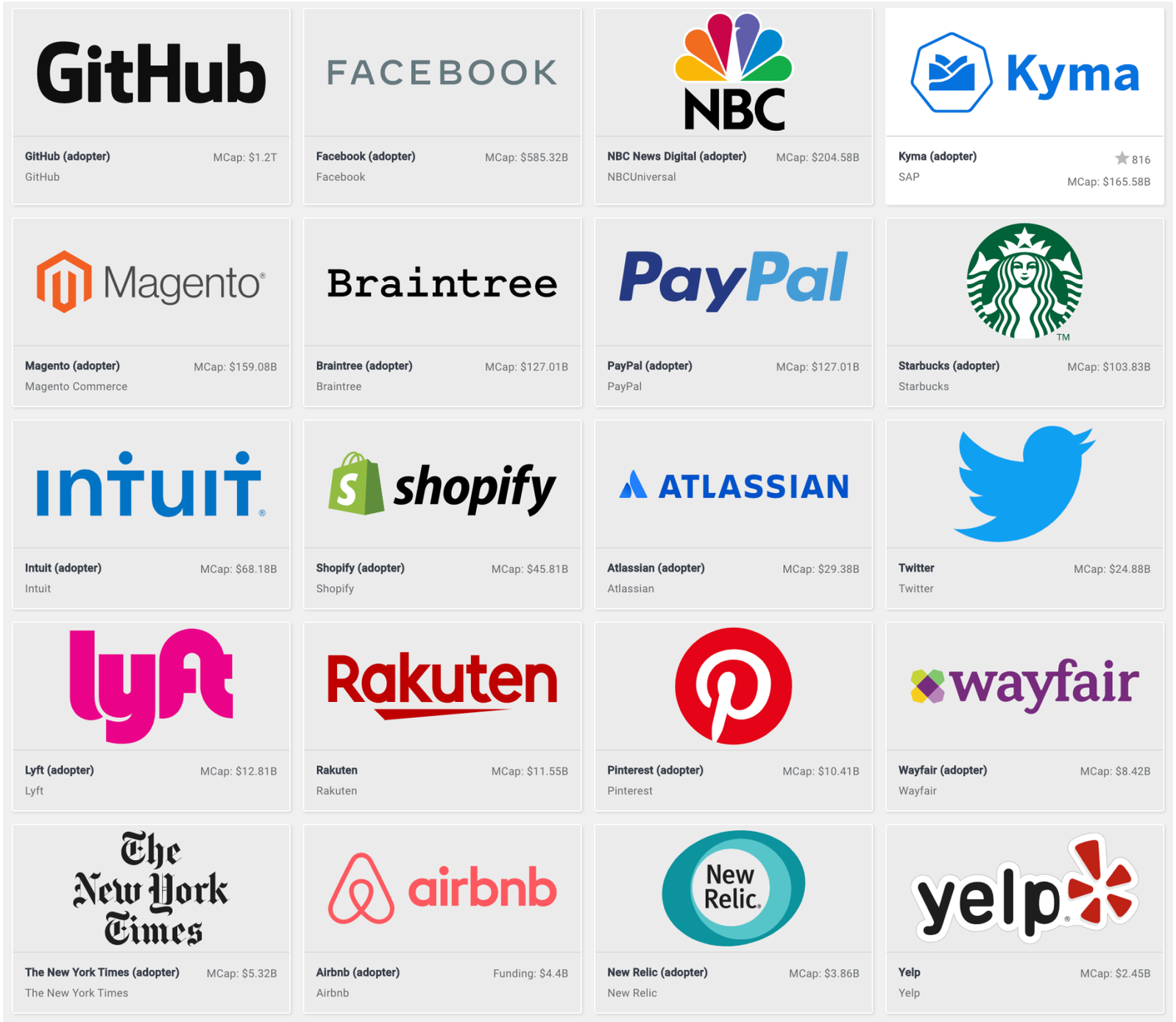
About Me

- Sr Software Engineer at Artsy
- We’re hiring! https://www.artsy.net/jobs
- https://github.com/damassi



GraphQL:

So, what's the hype?



https://landscape.graphql.org

The problem:

“When we took a closer look, we found that UI developers were spending less than 1/3 of their time actually building UI.”

“The rest of that time spent was figuring out where and how to fetch data, filtering/mapping over that data and orchestrating many API calls. Sprinkle in some build/deploy overhead. Now, building UI is a nice-to-have or an afterthought.”

— GraphQL: A success story for PayPal Engineering

Multiply that complexity * n number of platform targets

There has to be a better way…

But first! A disclaimer:

GraphQL is a vast and evolving subject and much was left out of this presentation

To learn more, check out:

https://graphql.org/

https://www.howtographql.com/

https://www.apollographql.com/

1:

TLDR - GraphQL in Practice

Client applications declare **exactly** what data they need via query, which maps 1:1 to its interface requirements (UI, etc)

```
query {
  artist(name: "Pablo Picasso") {
    biography
    yearsActive
    artworks {
      description
      createDate
      relatedArtworks {
        artist
        artwork {
          href
        }
      }
    }
  }
}
```

The GraphQL backend then returns **exactly** that data in the form of JSON

The backend, which consists of a schema, is an idealized “user interface” for your data layer

You design it, and it's **flexible.**

It can be anything you want, and connect to anything you want — internal databases, REST endpoints, arbitrary static data, etc.

This inverts things, compared to common REST approaches

```
GET /users
GET /artwork/2
POST /authors
```

The server no longer determines what is returned…

```
// /users
{
  ... tons of data ...
}

// artwork/2
{
  ... tons of data
}

// /authors
{
  ... tons of data ...
}
```

…requiring each client (and codebase) to then manipulate the response to fit the needs of the UI

No more of this, littered everywhere:

```
async function getFullName() {
  const user = await fetch('/user/2')
  return user.firstName + " " + user.lastName
}
```

Or a more commonly complex example:

```
async function fetchArtist() {
  const artist = await fetch('/artist/pablo-picasso')
  const artistArtworks = await fetch('/artist/pablo-picasso/artworks')
  const artworks = artistArtworks.map(async (artwork) => {
    const relatedArtworks = await fetch(`/related/${artwork.id}`)
    artwork.relatedArtworks = relatedArtworks
    return artwork
  })
  return {
    biography: artist.biography,
    artworks
  }
}
```

Logic that is often duplicated across codebases (desktop, mobile, web app, etc) is moved to a single location, the GraphQL server

artsy / metaphysics

Code | Issues 48 | Pull requests 6 | Actions

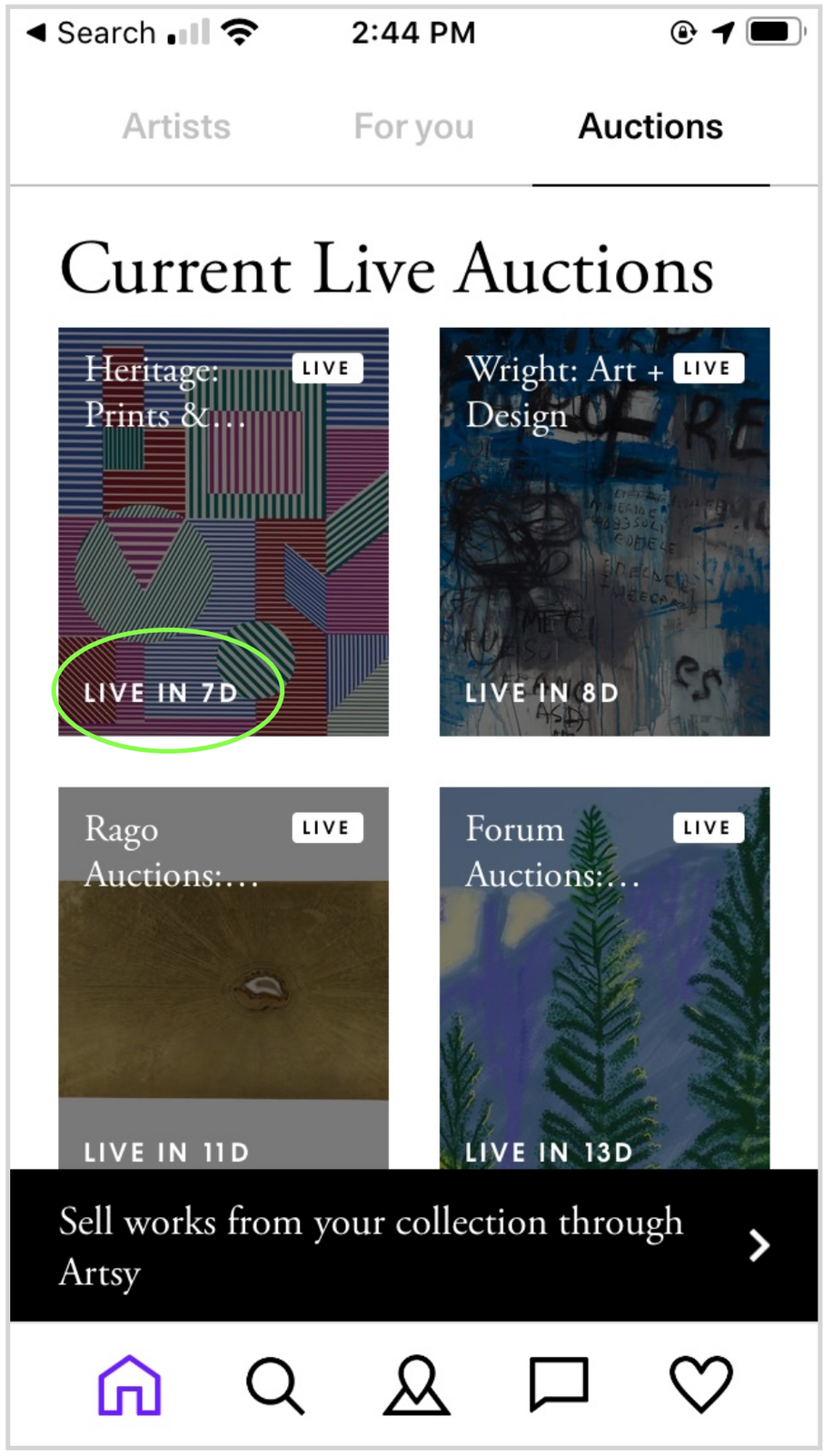
Branch: master — metaphysics / src / schema / v2 / sale /

```
displayTimelyAt: {
  type: GraphQLString,
  resolve: (sale, _options, { meBiddersLoader }) => {
    return displayTimelyAt({ sale, meBiddersLoader })
  },
},
```

GraphiQL ▶ Prettify | Merge | Copy | History

```
{
  salesConnection(first: 4) {
    edges {
      node {
        displayTimelyAt
      }
    }
  }
}
```

```
{
  "data": {
    "salesConnection": {
      "edges": [
        {
          "node": {
            "displayTimelyAt": "live in 11d"
          }
        },
        {
          "node": {
            "displayTimelyAt": "ends in 8d"
          }
        },
        {
          "node": {
            "displayTimelyAt": "live in 8d"
          }
        },
        {
          "node": {
            "displayTimelyAt": "live in 2Y"
          }
        }
      ]
    },
```



Client apps become nothing more than simple data containers encompassing **behavior**

```
function ArtistApp() {
  const data = useQuery(`
    artist(name: "Pablo Picasso") {
      fullName
      biography
      yearsActive
    }
  `)
  return (
    <div>
      <h1>{data.fullName}</h1>
      <p>{data.biography}</p>
      <div>{data.yearsActive}</div>
      <ul>
        {data.artworks.map(artwork => (
          <li>
            <strong>{artwork.description}</strong>
            <p>{artwork.createDate}</p>
          </li>
          )
        )}
      </ul>
    </div>
  )
}
```

As a product developer, behavior is what you want to work on, not figuring out how to fetch and display the same data across a handful of platform targets

2:

Origins

Facebook started a rebuild of their mobile iOS newsfeed

It… didn’t go well.

In computer programming and software engineering, the ninety-ninety rule is a humorous aphorism that states: The first **90 percent** of the code accounts for the first **90 percent** of the development time. The remaining **10 percent** of the code accounts for the other **90 percent** of the development time.

Ninety-ninety rule - Wikipedia

https://en.wikipedia.org › wiki › Ninety-ninety_rule

Lee Byron (@leeb)

Co-creator of GraphQL and
Executive Director of the GraphQL
Foundation

Nick Schrock (@schrockn)

Co-creator of GraphQL

“[We] got to work trying to figure out how to build a better News Feed API and we just got super far down the rabbit hole”

Facebook is a **Very Evil** **Company** but it knows rabbit holes well

Their internal engineering tools are typically a few years ahead of everyone else’s, and when open sourced are often revolutionary

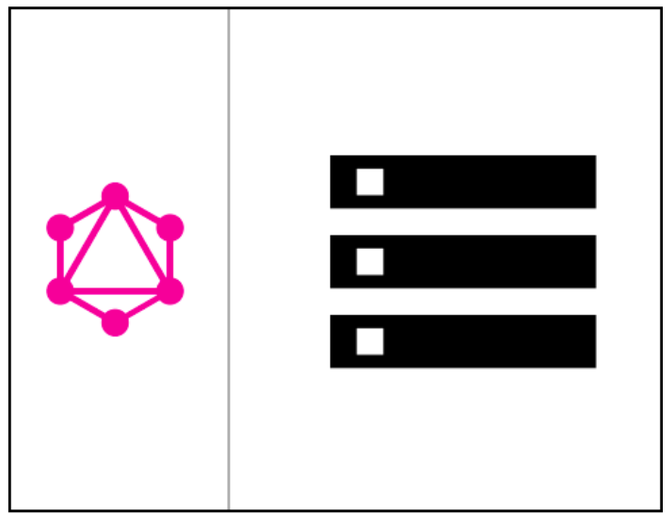
3:

TLDR - GraphQL in Theory

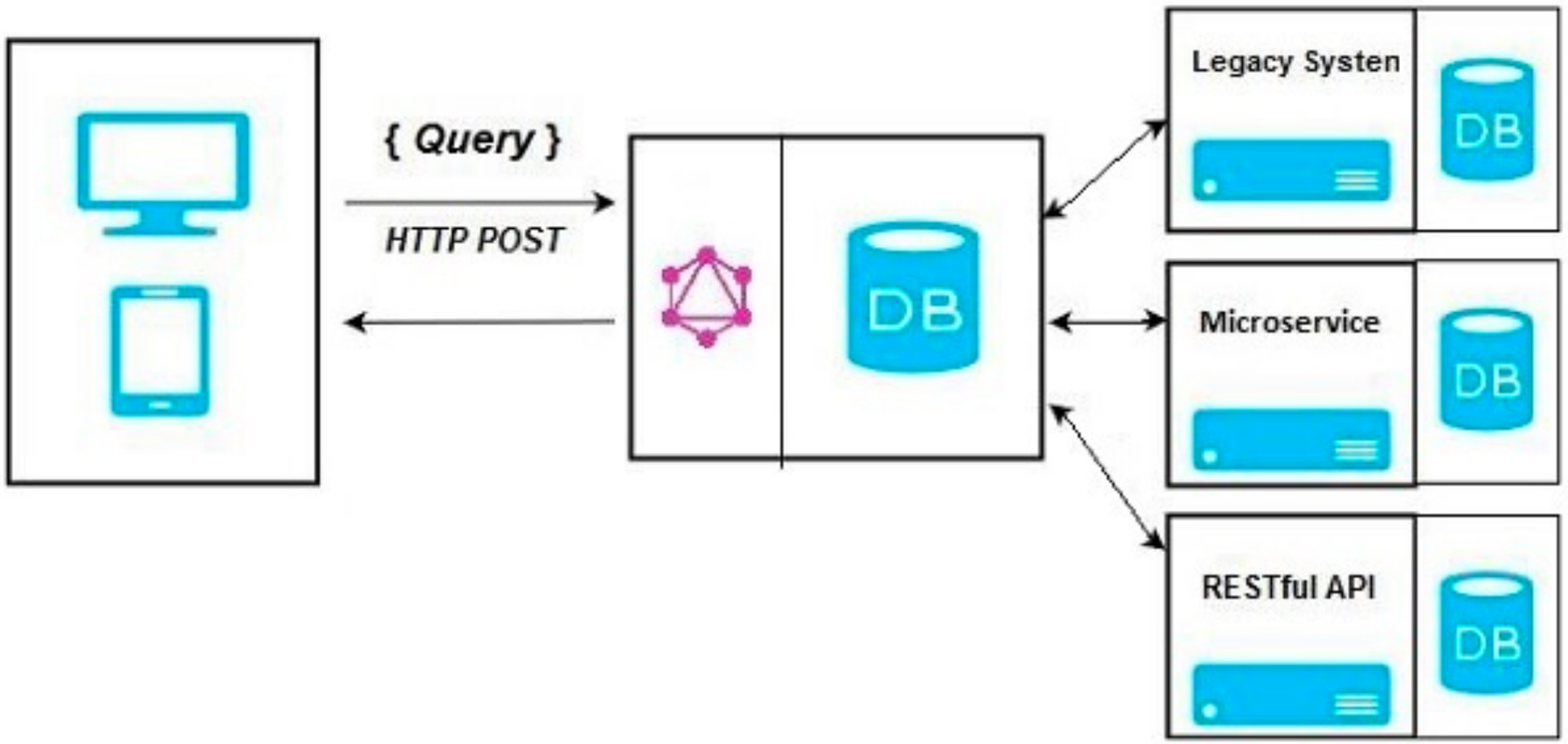
a) It’s a query language for your API

```
query {
  User(id: “er3tg439frjw”) {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```

HTTP POST



```
{
  “data”: {
    “User”: {
      “name”: “Mary”,
      “posts”: [
        { title: “Learn GraphQL today” }
      ],
      “followers”: [
        { name: “John” },
        { name: “Alice” },
        { name: “Sarah” },
      ]
    }
  }
}
```


{ Query }
HTTP POST
DB
Legacy Systen
DB
Microservice
DB
RESTful API
DB

b) It’s a specification independent of language or framework

https://graphql.github.io/graphql-spec

Server languages then implement the spec in the form of libraries

C# / .NET, Clojure, Elixir, Erlang,
Go, Groovy, Java, JavaScript,
Kotlin, PHP, Python, Ruby, Rust
Scala, Swift — and more

c) It’s a Type System

```
{
  hero {
    name
    friends {
      name
      homeWorld {
        name
        climate
      }
      species {
        name
        lifespan
        origin {
          name
        }
      }
    }
  }
}
```

```
type Query {
  hero: Character
}

type Character {
  name: String
  friends: [Character]
  homeWorld: Planet
  species: Species
}

type Planet {
  name: String
  climate: String
}

type Species {
  name: String
  lifespan: Int
  origin: Planet
}
```

Because of its type system, sophisticated developer tooling and self-generating documentation can easily be created

GitHub GraphQL API

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

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

GraphiQL ▶ Prettify History

```
{
  repository(owner: "umermansoor", name: "microservices") {
    name
    nameWithOwner
    description
    forkCount
    # This is a nested object
    stargazers {
      totalCount
    }
  }
}
```

QUERY VARIABLES

```
{}
```

```
{
  "data": {
    "repository": {
      "name": "microservices",
      "nameWithOwner": "umermansoor/microservices",
      "description": "Example of Microservices written using Flask.",
      "forkCount": 199,
      "stargazers": {
        "totalCount": 972
      }
    }
  }
}
```

Schema **Repository**

description: String
The description of the repository.

descriptionHTML: HTML!
The description of the repository rendered to HTML.

diskUsage: Int
The number of kilobytes this repository occupies on disk.

forkCount: Int!
Returns how many forks there are of this repository in the whole network.

forks(
  privacy: RepositoryPrivacy
  orderBy: RepositoryOrder
  affiliations: [RepositoryAffiliation] = [OWNER, COLLABORATOR]
  ownerAffiliations: [RepositoryAffiliation] = [OWNER, COLLABORATOR]
  isLocked: Boolean
  after: String
  before: String
  first: Int
  last: Int

d) An ecosystem of client libraries used to communicate with a GraphQL-enabled server

Relay

https://relay.dev/

APOLLO

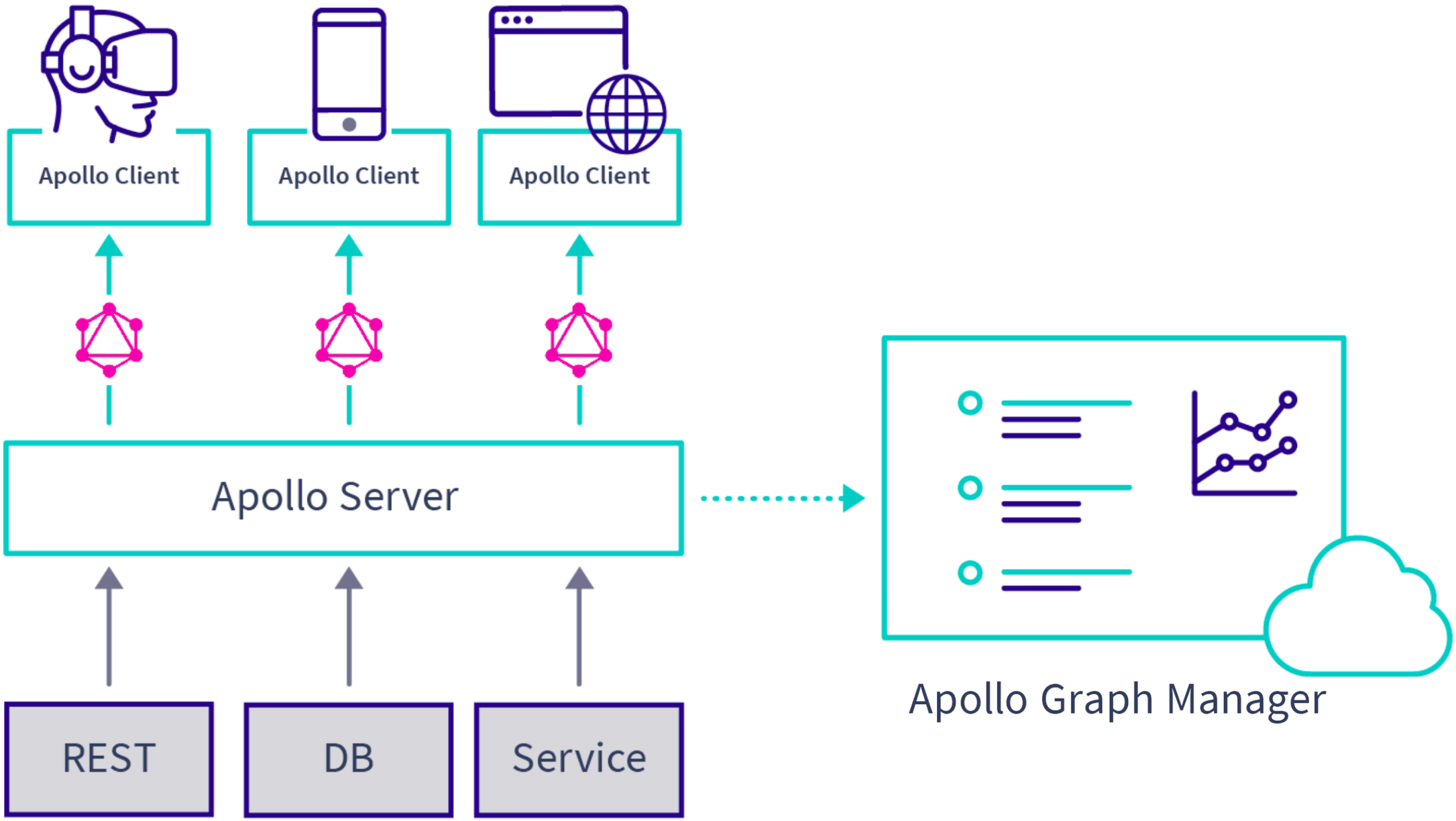
https://www.apollographql.com/

Artemis

https://github.com/yuki24/artemis

```
import { useQuery } from '@apollo/react-hooks'

function ArtistApp() {
  const data = useQuery(`
    artist(name: "Pablo Picasso") {
      fullName
      biography
      yearsActive
    }
  `)
  return (
    <div>
      <h1>{data.fullName}</h1>
      <p>{data.biography}</p>
      <div>{data.yearsActive}</div>
    </div>
  )
}
```


Apollo Client
Apollo Client
Apollo Client
Apollo Server
REST
DB
Service
Apollo Graph Manager

4:

Designing a GraphQL server

But first, an (incomplete) glossary of terms

A GraphQL server consists of:

Type definitions (SDL)

Resolvers, which map to types and return data

Type definitions

Given a DATABASE consisting of a TABLE of users*…

```
// Example DB table
{
  "users": [
    {
      "username": "hello",
      "id": 0
    },
    {
      "username": "world",
      "id": 1
    }
  ]
}
```

One creates a type definition…

```
type User {
  username: String
  id: Int
}

type Query {
  users: [User]
}
```

*** Doesn't *have* to be a database; used as a quick example to model a common data structure**

Resolvers map to types and return data

```
const resolvers = {
  Query: {
    users: async () => {
      const users = await UsersTable.getAll()
      return users
    },
  },
}
```

</glossary>

Typically, there are three possible server configurations

a) GraphQL Monolith - No underlying API; schema speaks directly to a DB

```
{
  query {
    users {
      username
      id
    }
  }
}
```

→

```
const typeDefs = gql`
  type User {
    username: String
    id: Int
  }

  type Query {
    users: [User]
  }
`

const resolvers = {
  Query: {
    users: async () => {
      const users = await UsersTable.getAll()
      return users
    },
  },
}
```

→

```
{
  "data": {
    "users": [
      {
        "username": "hello",
        "id": 0
      },
      {
        "username": "world",
        "id": 1
      }
    ]
  }
}
```

b) API Gateway - GraphQL server is used to communicate with other services and consolidate responses behind single endpoint

Slightly more complex example…

```graphql
const typeDefs = gql`
  type Query {
    authors: [Author]
    books: [Book]
  }

  type Author {
    id: String
    firstName: String
    lastName: String
    fullName: String
    books: [Book]
  }

  type Book {
    id: String
    author: Author
    title: String
  }
`
```

→

```javascript
const resolvers = {
  Query: {
    authors: async () => {
      const authors = await fetch('https://authors-service.com/api/authors')
      return authors
    },
    books: async () => {
      const books = await fetch('https://books-service.com/books/api/books')
      return books
    }
  },
  Author: {
    fullName: author => {
      return author.firstName + ' ' + author.lastName
    },
    books: author => {
      const authorBooks = await fetch(`https://books-service.com/books/api/books?authorID=${author.id}`)
      return authorBooks
    }
  },
  Books: {
    author: book => {
      const bookAuthor = await fetch(`https://authors-service.com/api/authors?bookID=${book.id}`)
      return bookAuthor
    }
  },
}
```

c) A combination of A and B

```
const resolvers = {
  Query: {
    authors: async () => {
      const authors = await fetch('https://authors-service.com/api/authors')
      return authors
    },
    books: async () => {
      const books = await BooksDB.getAll()
      return books
    }
  },
  Author: {
    fullName: author => {
      return author.firstName + ' ' + author.lastName
    },
    books: author => {
      const authorBooks = await BookDB.where({
        authorID: author.id
      })

      return authorBooks
    }
  },
}
```

Q:

All of this seems possible with <x, y, z>, why are we reinventing the wheel?

A:

We're not. We're creating an idealized version of our data layer using a pattern that enables the creation of idealized forms, thanks to flexibility

And crucially, this pattern scales well, at a minimal upfront setup cost*

***Assuming you've done your research!**

5:

Whats the catch?

The examples in the previous slides are entirely MVP (minimum viable product)

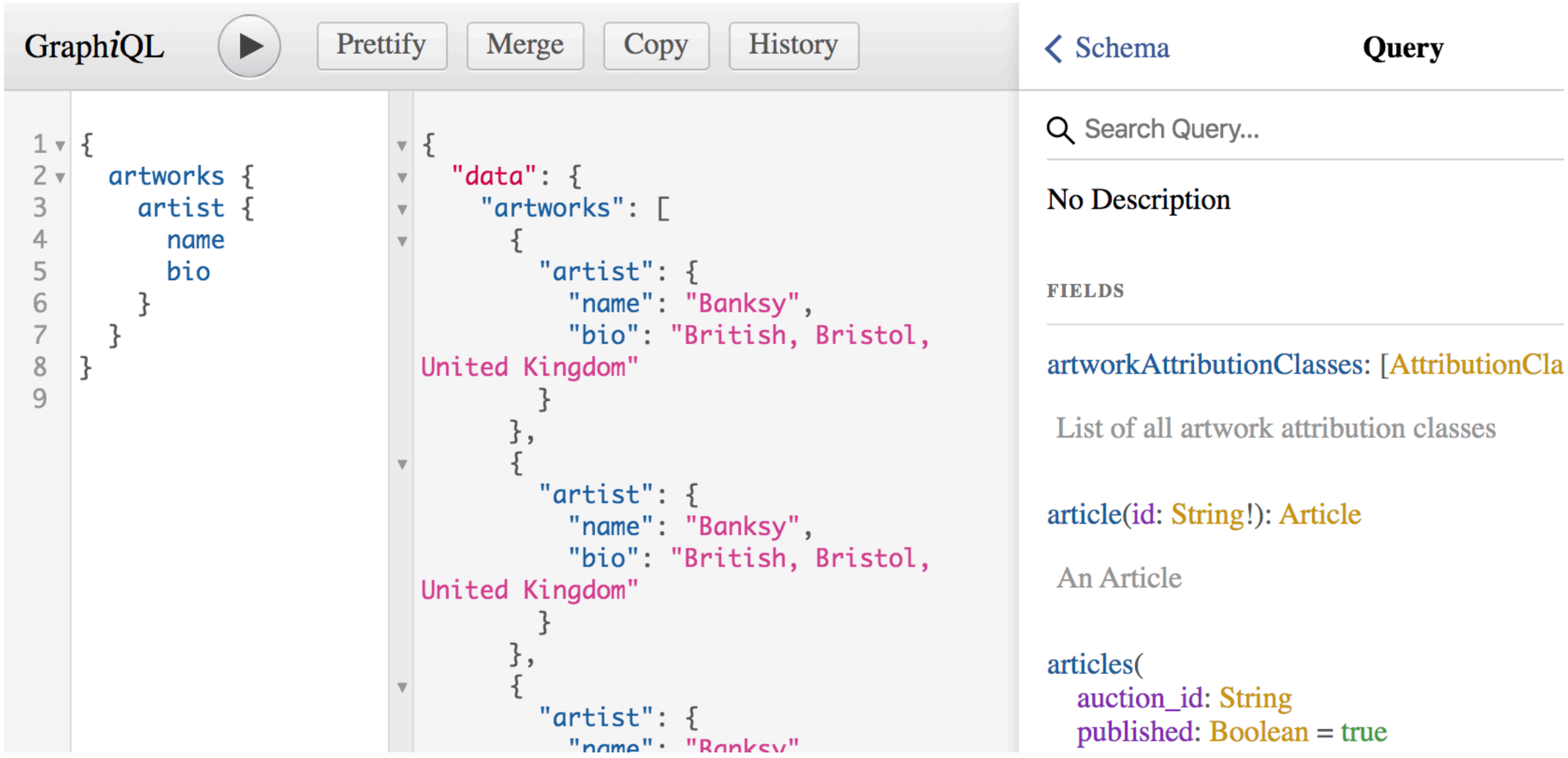
Developers need to be aware of common pitfalls

a) Thinking cap is absolutely required. There's overhead and learning involved

However: I've **worked** with GraphQL in client-side applications for much longer than I've *understood* it.

As a developer on a team building UI this is (basically) it:

Look up the data you need in GraphiQL

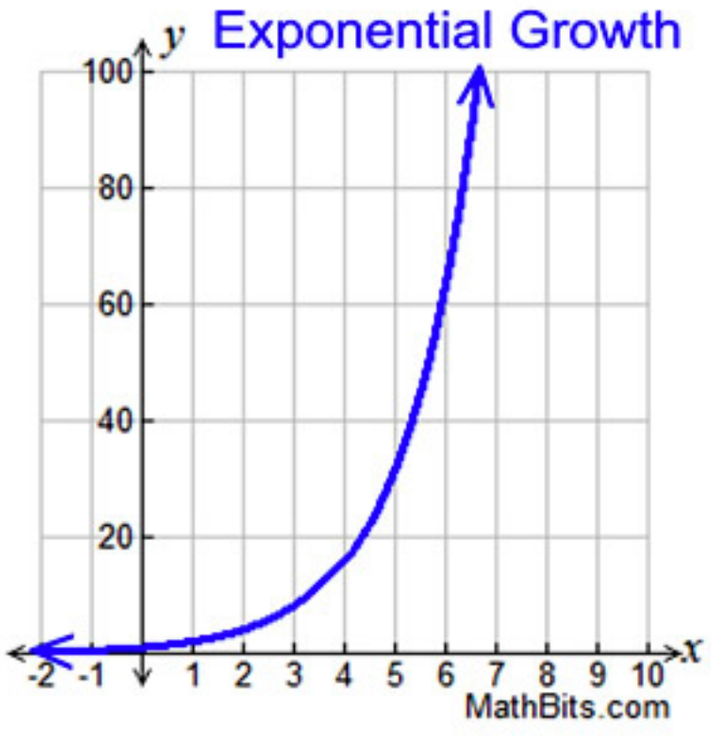


Copy the query, paste it into your component

```
function MyApp() {
  const data = useQuery(`
    query {
      artworks {
        artist {
          name
          bio
        }
      }
    }
  `)

  return (
    <div>
      {data.artworks.map(artwork => (
        <div>
          <h1>{artwork.artist.name}</h1>
          <p>{artwork.artist.bio}</p>
        </div>
      ))}
    </div>
  )
}
```

Done. Now, get to work making that component look good and function real well :)



(Engineering team velocity after adopting GraphQL)

But at a certain point one will be expected to add new fields to their schema backend, or adjust existing ones.

This requires understanding.

b) Performance issues with complex queries

```
query {
  user {
    favorites {
      books {
        author {
          related {
            books
          }
        }
      }
    }
  }
}
```

"I'd like a list of books related to current user's favorite authors' books”

This query is 5 levels deep, which means potentially five underlying requests

GraphQL is just a query **language.** It resolves data in an agnostic way. If your datastore already has problems, GraphQL will not fix it.

(Think about rate limiting and other measures just to be safe!)

b) N+1 requests, caching and batching

“It is *much faster* to issue 1 query which returns 100 results than to issue 100 queries which each return 1 result”

Bad...

```
SELECT * FROM cat WHERE ...
SELECT * FROM hat WHERE catID = 1
SELECT * FROM hat WHERE catID = 2
SELECT * FROM hat WHERE catID = 3
SELECT * FROM hat WHERE catID = 4
SELECT * FROM hat WHERE catID = 5
...
```

Good!

```
SELECT * FROM cat WHERE ...
SELECT * FROM hat WHERE catID IN (1, 2, 3, 4, 5, ...)
```

- https://github.com/graphql/dataloader
    - DataLoader is a generic utility to be used as part of your application's data fetching layer to provide a consistent API over various backends and reduce requests to those backends via batching and caching.
- https://www.npmjs.com/package/apollo-datasource-rest
    - Data sources are classes that encapsulate fetching data from a particular service, with built-in support for caching, deduplication, and error handling.
- https://github.com/Shopify/graphql-batch
    - Provides an executor for the graphql gem which allows queries to be batched.

6: Live code time!

Example application showing how to build a full-stack server/client GraphQL app via Apollo, React, and TypeScript

https://github.com/damassi/graphql-by-example

Artsy's GraphQL server. It's pretty large with a lot of patterns one can explore

https://github.com/artsy/metaphysics

Thanks!

(And extra thanks to all of the creators behind misc quotes / images I plucked from google and used in this presentation.)