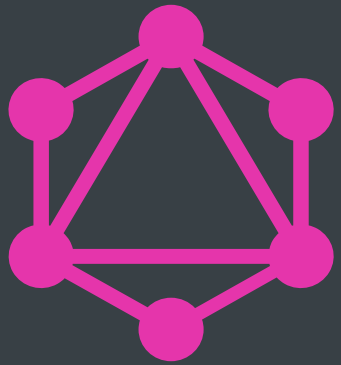


Introduction to GraphQL

Niv Ben David

Agenda

```
graphql {
    why
    what
    how
}
```

History, Context & Adoption

- GraphQL is an open source query language created by Facebook.
- Before GraphQL went open source in 2015, Facebook used it internally for their mobile applications since 2012, as an alternative to the common REST architecture.
- Facebook open-sourced the GraphQL specification and its reference implementation in JavaScript, and multiple major programming languages have implemented the specification since then.
- The ecosystem around GraphQL is growing horizontally by offering multiple programming languages, but also vertically, with libraries on top of GraphQL like Apollo and Relay.
- The first time Facebook publicly spoke about GraphQL was at React.js Conf 2015 and shortly after announced their plans to open source it. Because Facebook always used to speak about GraphQL in the context of React, it took a while for non-React developers to understand that GraphQL was by no means a technology that was limited to usage with React.

A rapidly growing Community

In fact, GraphQL is a technology that can be used everywhere a client communicates with an API.

Interestingly, other companies like Netflix or Coursera were working on comparable ideas to make API interactions more efficient.

Coursera envisioned a similar technology to let a client specify its data requirements and Netflix even open-sourced their solution called Falcor. After GraphQL was open-sourced, Coursera completely cancelled their own efforts and hopped on the GraphQL train.

Today, GraphQL is used in production by lots of different companies such as GitHub, Twitter, Yelp, Paypal and Shopify - to name only a few.

There are entire conferences dedicated to GraphQL such as GraphQL Conf and more resources like the GraphQL Weekly newsletter.

A more efficient Alternative to REST

REST has been a popular way to expose data from a server.

When the concept of REST was developed, client applications were relatively simple and the development pace wasn’t nearly where it is today.

REST thus was a good fit for many applications.

However, the API landscape has radically changed over the last couple of years.

In particular, there are three factors that have been challenging the way APIs are designed:

1. Increased mobile usage creates need for efficient data loading

Increased mobile usage, low-powered devices and sloppy networks were the initial reasons why Facebook developed GraphQL. GraphQL minimizes the amount of data that needs to be transferred over the network and thus majorly improves applications operating under these conditions.

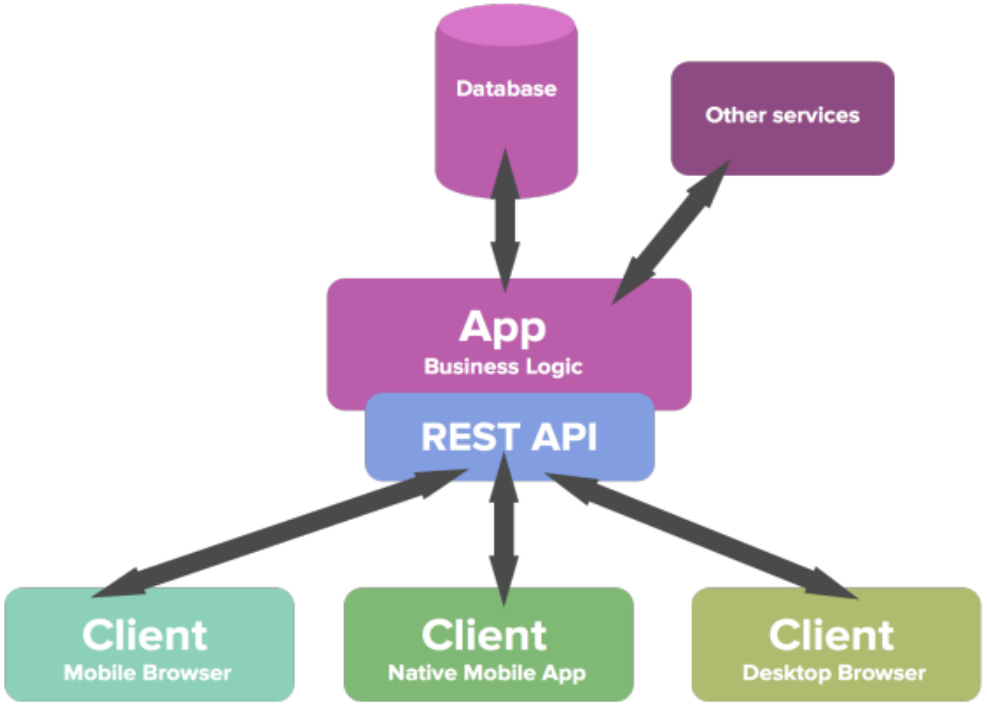
2. Variety of different frontend frameworks and platforms

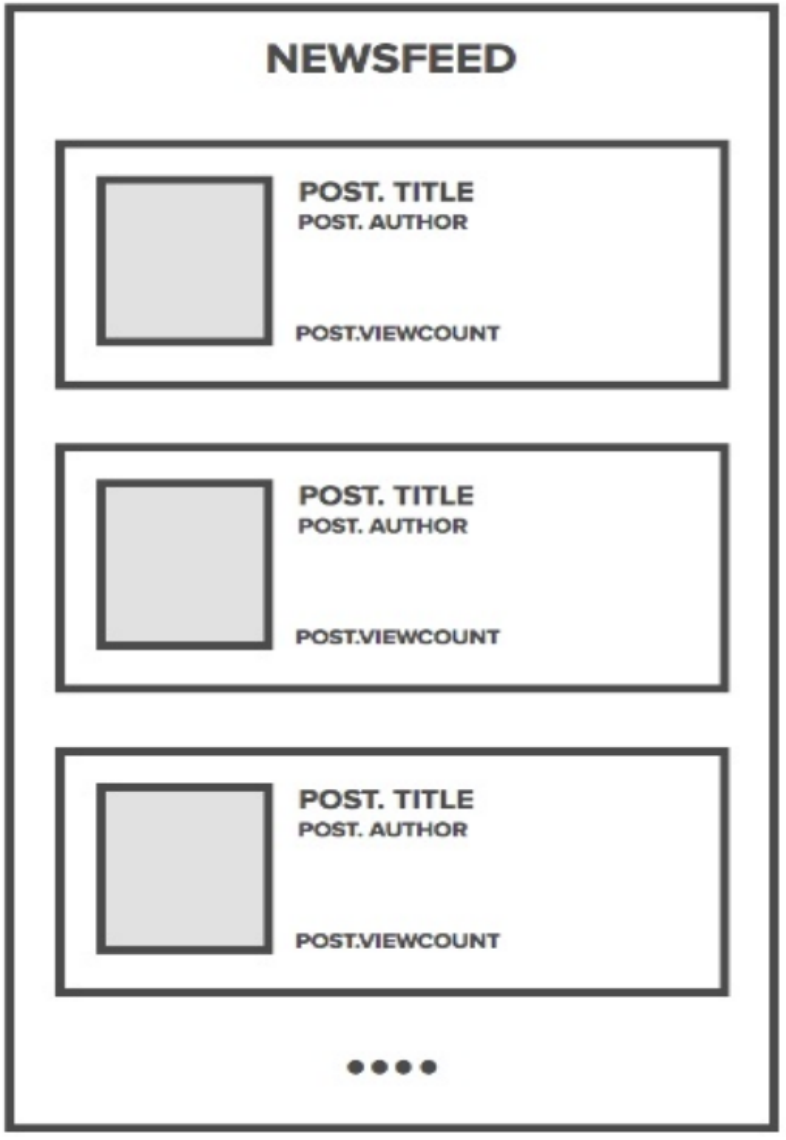
The heterogeneous landscape of frontend frameworks and platforms that run client applications makes it difficult to build and maintain one API that would fit the requirements of all. With GraphQL, each client can access precisely the data it needs.

3. Fast development & expectation for rapid feature development

Continuous deployment has become a standard for many companies, rapid iterations and frequent product updates are indispensable. With REST APIs, the way data is exposed by the server often needs to be modified to account for specific requirements and design changes on the client-side. This hinders fast development practices and product iterations.

Typical architecture of a web application using REST API

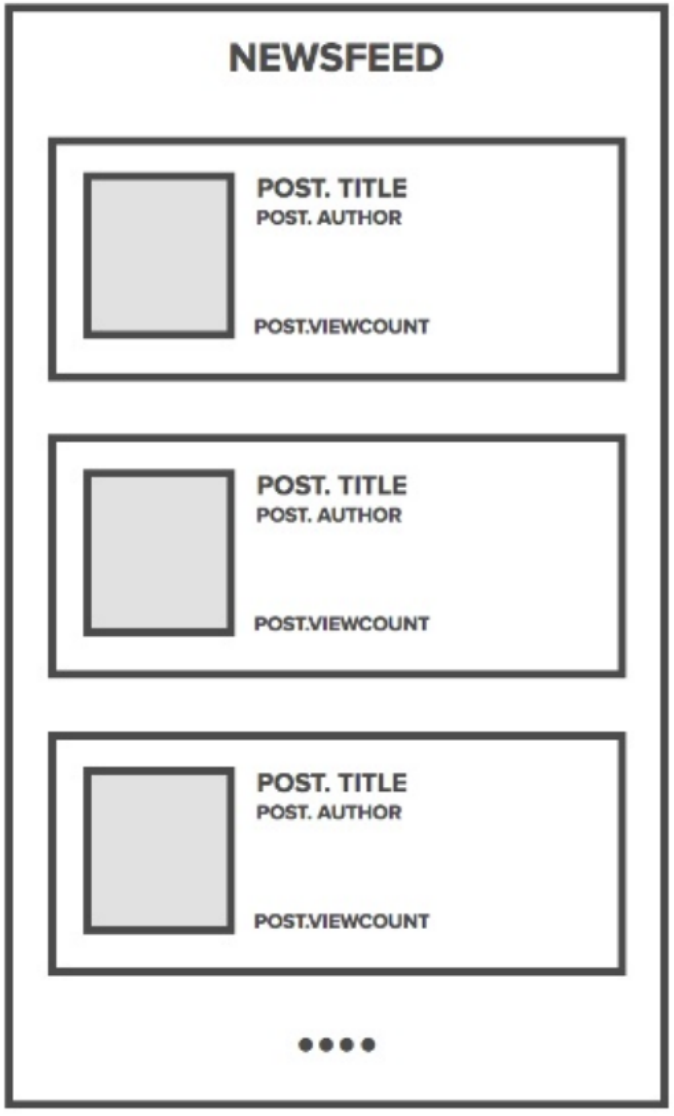




Goal: A newsfeed SPA

Mobile-first

REST API for data fetching

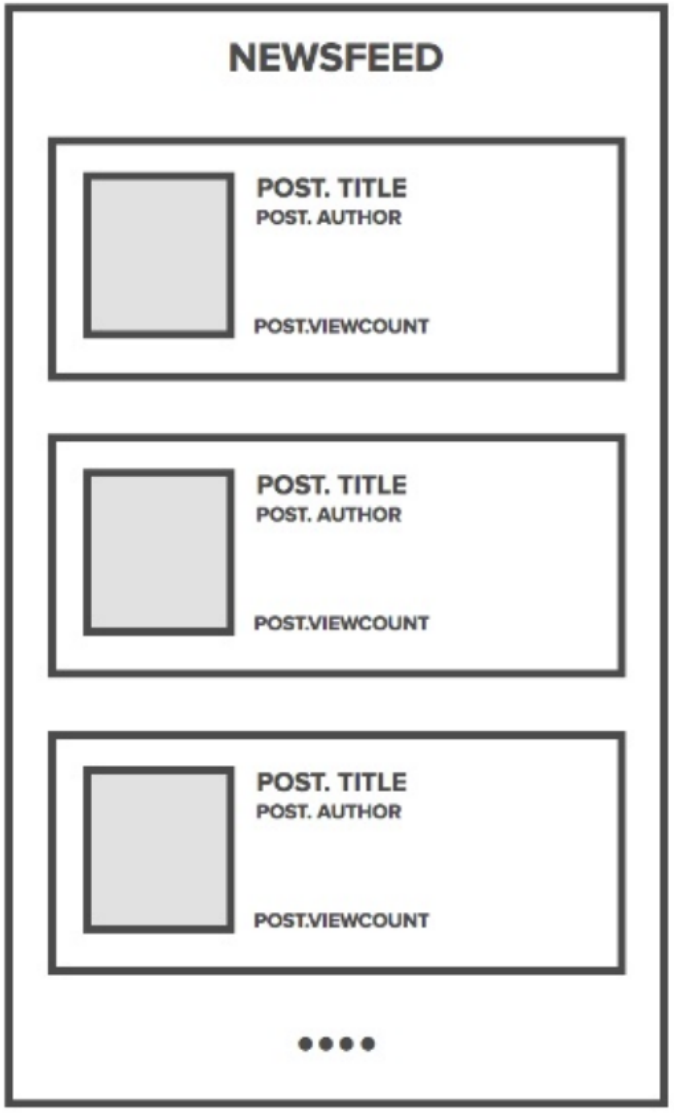


Designing the REST API

Two **resources**

- Users
- Posts

```
POST /posts
GET /posts/1
PUT /posts/1
DELETE /posts/1
...
POST /users
GET /users/1
PUT /users/1
DELETE /users/1
...
```



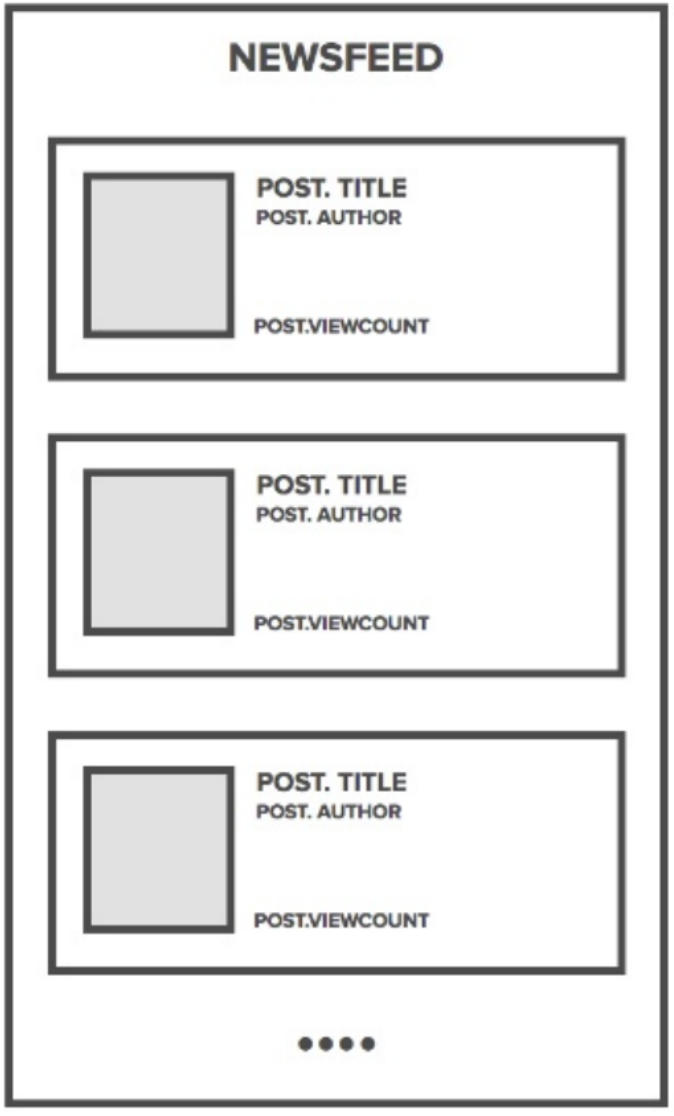
Render newsfeed

```
GET /posts?limit=10
{
  "posts": [
    {
        "id": 1,
        "title": "Hello world!",
        "author": 10,
        "viewCount": 23,
        "likedCount": 3,
        "likedBy": [1, 3],
    },
    ...
  ]
}

-
```

Great!

Oh wait, we need to get **`author's name`** and **`avatar URL`**

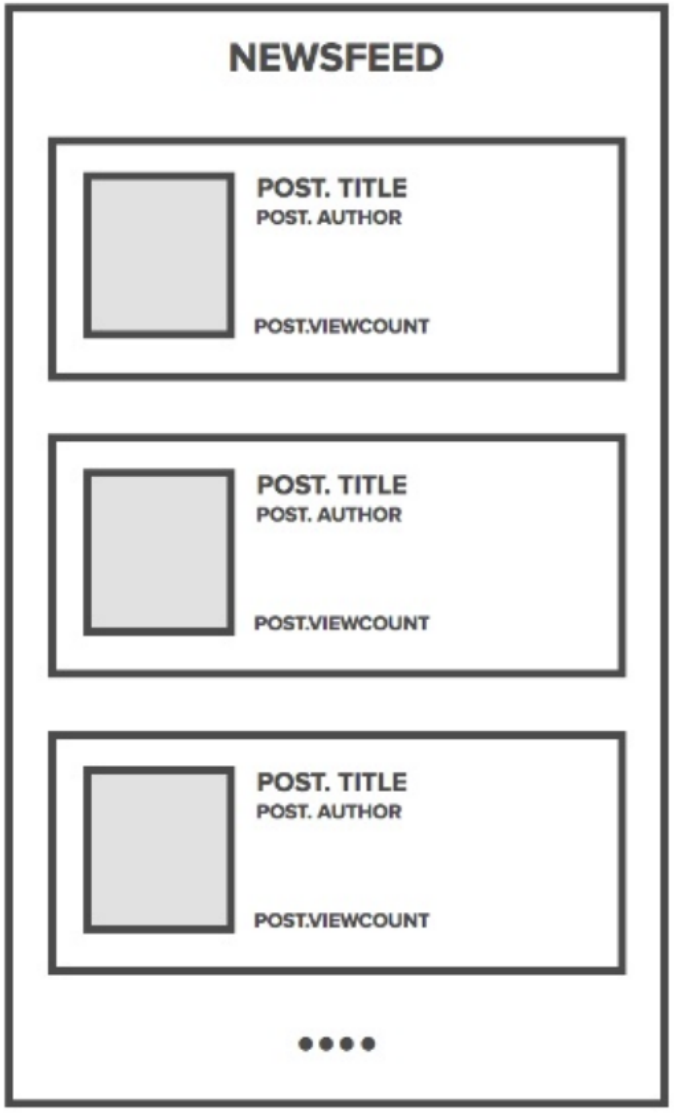


Render newsfeed

```
GET /posts?limit=10
GET /users/10
{
  "user": {
        "id": 10,
        "name": "John Doe",
        "nickname": "Johnny",
        "age": 23,
        "avatar_url": "/avatar/10.jpg"
    }
}

_
```

So we make another request to get the author for the first post...



Render newsfeed

```
GET /posts?limit=10
GET /users/10
GET /users/20
{
  "user": {
        "id": 20,
        "name": "Emily Sue",
        "nickname": "M",
        "age": 25,
        "avatar_url": "/avatar/20.jpg"
    }
}
```

Wait, so we have to do a separate request for each post to get information for its author?

Hhnnggghhh 😠

Issue #1: Multiple round trips

Issue #1: Multiple round trips

One possible solution:
- A new endpoint **`/newsfeed`**

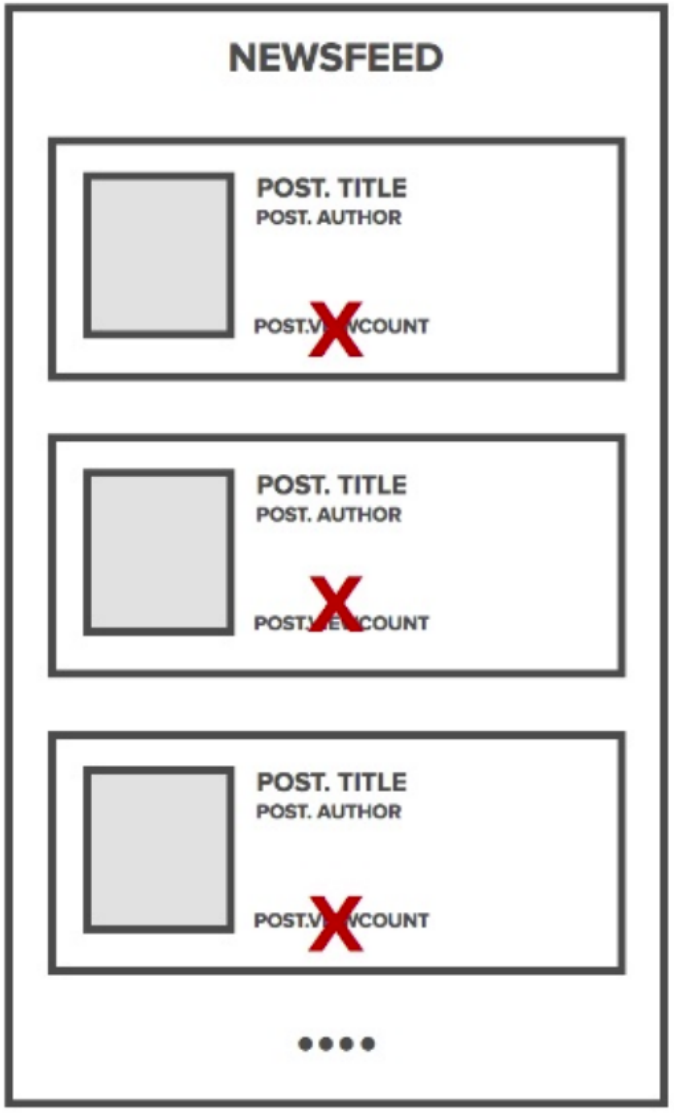
But now you have a singleton REST resource that is too tightly-coupled to your client UI.

You then tell yourself "it's not that bad. We're still REST-ish."

Eventually, you launch your app with its mobile client and it went viral! Yay!

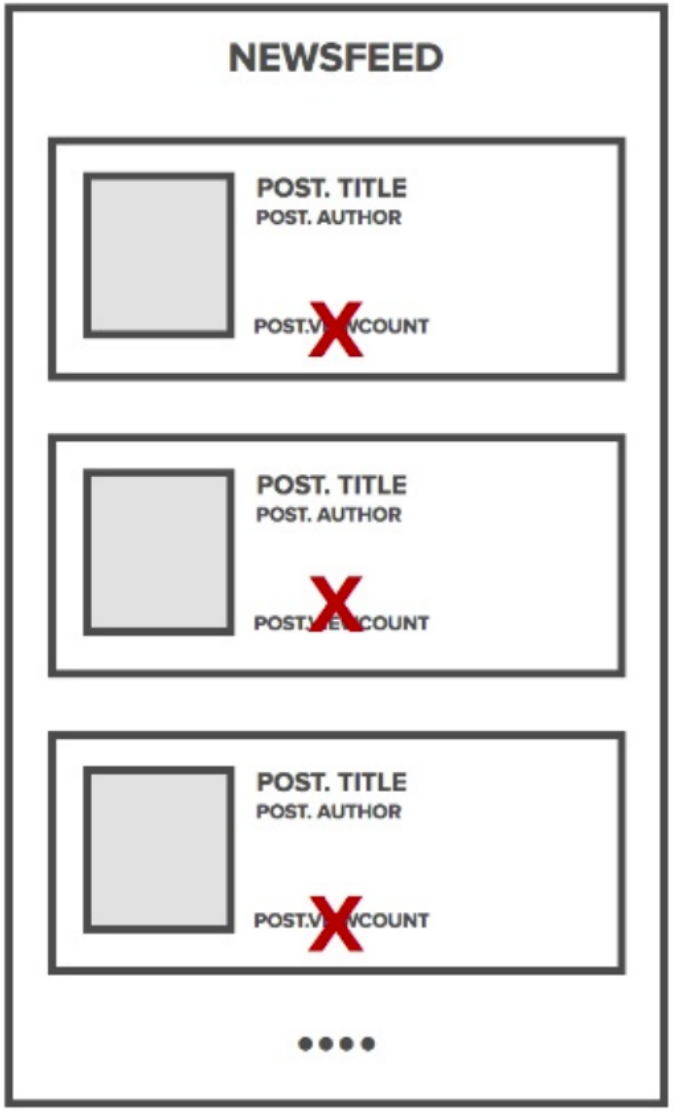
New requirement!

Here comes your product designer with changes

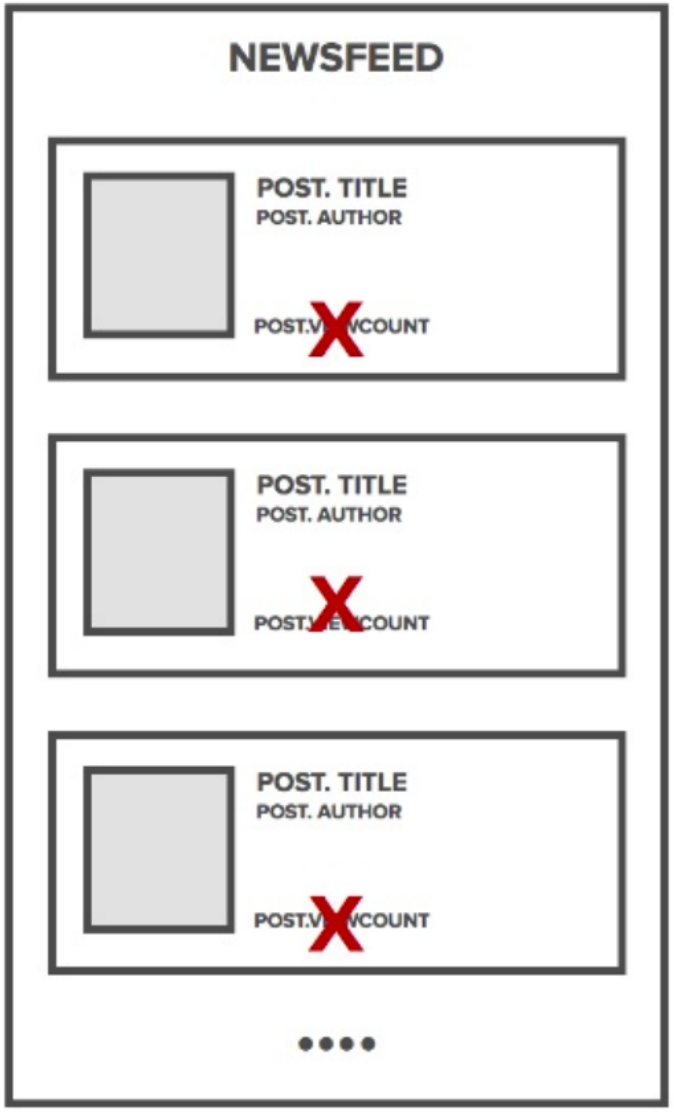


New requirement!

- It has been a year since you first launch your mobile client, and you have several versions of the client out in the wild.
- Your product designer said: "We need to stop showing the `view_count` because of reasons"
- What do you do now?



- Removing the **`view_count`** field from **`/newsfeed`** is not an option.
- Older version of your mobile client depends on it. (What if they crash? #nilpointerreference)
- So newer version of the mobile client **does not need** the **`view_count`** field, but to cater to the older versions, the **`/newsfeed`** still need to return it.



- **What if this keeps happening?** Newer clients would be requesting data that they essentially don't need anymore.
- Not that bad when you just start out, but in the long run, it'll be something nagging at you
- Sleepless nights are ahead for you.

Issue #2:
Overfetching of data

Issue #2: Overfetching of data

Wouldn't it be nice if there client receives only the data that it requires and had requested?

One possible way to go about this:

- Endpoint accepts parameter to specify the fields that you are interested in

Not a bad solution, but *yet* another thing to worry about.

From a humble SPA to a full-fledge product

You and your core team have build this complex API that serves a great purpose.

From a humble SPA to a full-fledge product

- Your CEO recently announced that he envisions that your product should have a **client for every device / platform imaginable**.
- iOS, Android, OSX, Windows, Linux
- Raspberry Pis, BBC micro:bits
- Cars
- Your mom's toaster

From a humble SPA to a full-fledge product

- New hires/developers join in.
- How do you quickly allow new developers to study your API
  - What resources are available,
  - What parameters are accepted
  - Which ones are required, which ones are not?

From a humble SPA to a full-fledge product

- If only you had used Swagger / RAML / API Blueprint when you started.
- Now you have to invest time/effort into it.
- Or probably you already did, bonus points for you 👍🏻

Issue #3: Documenting your API now becomes a thing.

Issue #3: Documenting your API now becomes a thing.

- More than just writing down the specs in a formal form so that it can be referenced
- How you allow one to **discover** and **explore** your API?
- Big enough concern that some parts of the community has banded together to create tools for this. *(Which is a great thing, yay open-source)*
- But *yet* another thing for you to worry about.

Had enough?

So how do we proceed from here?

What is GraphQL?

- GraphQL is just a web **protocol** that specifies the way we build and query remote APIs using a Tree/JSON like syntax.

```
reactorHubNews {
    title
    description
}
```

- Based on a strongly **typed** language - GraphQL Schema Definition Language (SDL).

```
type Post {
  title: String!
  description: String!
}
```

What is GraphQL?

- A protocol that allows the client to specify exactly **what data it needs** from a model.

```
posts {
   title
}
```

- It allows to aggregate data from **multiple relations** in a single query.

```
posts {
   title
   description
   user {
      name
   }
}
```

GraphQL Type System - Built in scalar types

- **Int** - An Integer type, example 10
- **Float** - Floating point number , example 3.43
- **String** - A sequence of characters , example “Hello World”
- **Boolean** -
- **ID** - Object identifier

GraphQL Type System - User Defined Types

- GraphQL uses types to ensure the clients know the fields supported by a resource. The types are defined by the user following the GraphQL SDL specification

```
type Post {
  title: String!
  description: String!
  author: User!
}

type User {
 fullName: String!
 email: String!
}
```

GraphQL Query

- Used to fetch data from the server using the GraphQL SDL syntax.
- Describe what data the requester wishes to fetch from whoever is fulfilling the GraphQL query.

```
query reactorNews {
   posts(sortedBy: createdDesc) {
      title
      description
       author {
        fullName
      }
   }
}
```

GraphQL Mutation

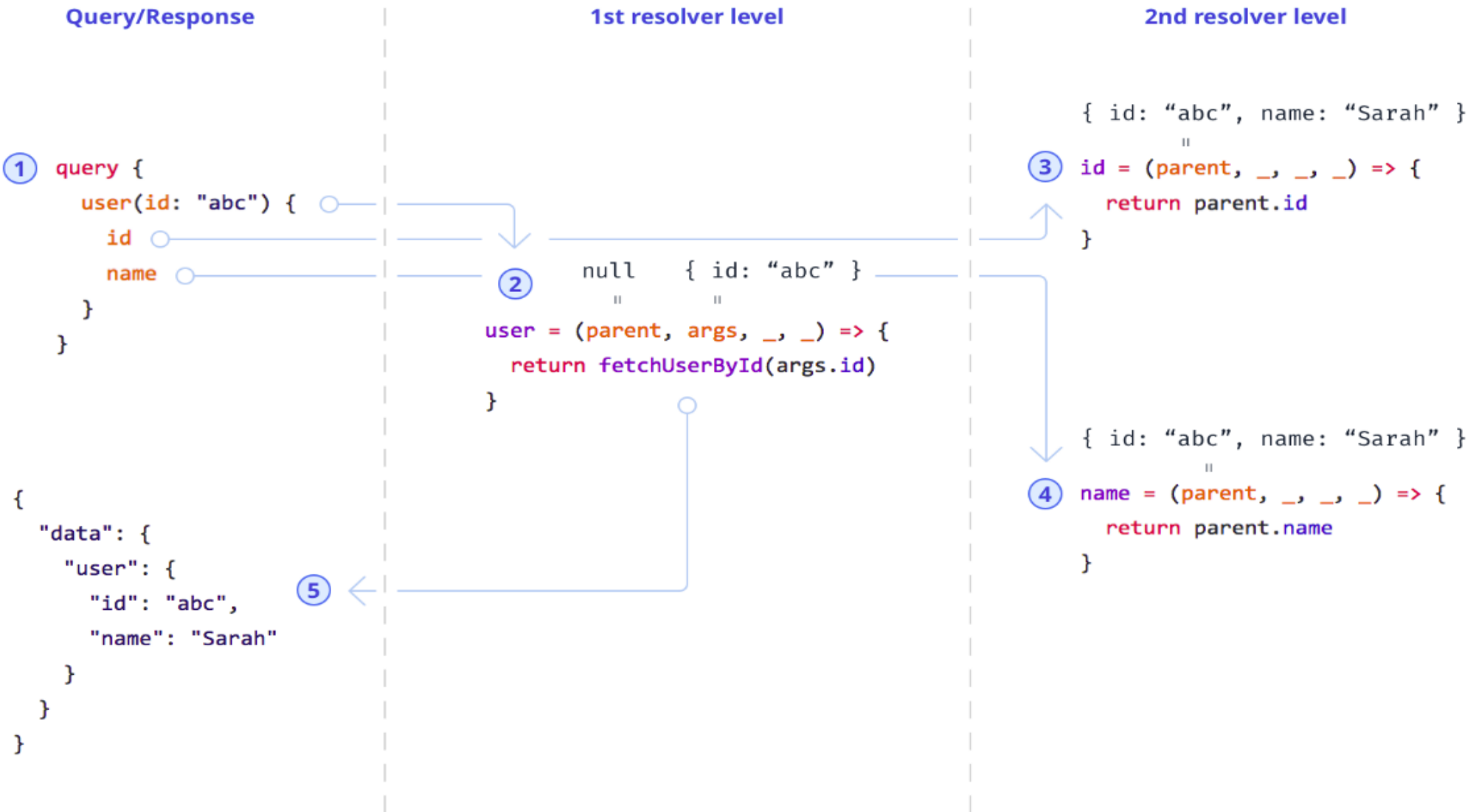
- Used to change resources data or execute actions on the server
- Client specifies the arguments and the action to be executed and at the end receives a response or a resource updated

```
mutation createPost {
   createPost({
      title: "Reactor is celebrating a party"
      Description: "Beer, sparkling water and much more….:)" })
  {
      id
      title
  }
}
```

GraphQL Subscription

- Used to get realtime resource or data updates
- Users get a notification every time the resource subscribed gets updated or changed.

```
subscription newSubscriber {
   newsLetterSubscriberCreated{
      id
      subscriber {
         fullName
         email
      }
      createAt
   }
}
```


Query/Response
1st resolver level
2nd resolver level
{ id: “abc”, name: “Sarah” }
1
query {
user(id: "abc") {
id
name
}
}
3
id = (parent, _, _, _) => {
return parent.id
}
2
null
{ id: “abc” }
user = (parent, args, _, _) => {
return fetchUserById(args.id)
}
{ id: “abc”, name: “Sarah” }
4
name = (parent, _, _, _) => {
return parent.name
}
5
{
"data": {
"user": {
"id": "abc",
"name": "Sarah"
}
}
}

GraphQL Advantages

- You get the data you request and need for a particular scope
- Excellent developer tooling and experiences since the specification defines API introspection
- Only one endpoint to connect,
    - `Example POST api.reactorhub.com /graphql/api`

- **No more Over- and Underfetching**

  One of the most common problems with REST is that of over- and underfetching. This happens because the only way for a client to download data is by hitting endpoints that return fixed data structures. It’s very difficult to design the API in a way that it’s able to provide clients with their exact data needs.

  Overfetching: Downloading superfluous data

  Overfetching means that a client downloads more information than is actually required in the app. Imagine for example a screen that needs to display a list of users only with their names. In a REST API, this app would usually hit the /users endpoint and receive a JSON array with user data. This response however might contain more info about the users that are returned, e.g. their birthdays or addresses - information that is useless for the client because it only needs to display the users’ names.

Underfetching and the n+1 problem

Another issue is underfetching and the n+1-requests problem. Underfetching generally means that a specific endpoint doesn’t provide enough of the required information. The client will have to make additional requests to fetch everything it needs. This can escalate to a situation where a client needs to first download a list of elements, but then needs to make one additional request per element to fetch the required data.

Rapid Product Iterations on the Frontend

A common pattern with REST APIs is to structure the endpoints according to the views that you have inside your app. This is handy since it allows for the client to get all required information for a particular view by simply accessing the corresponding endpoint.

The major drawback of this approach is that it doesn’t allow for rapid iterations on the frontend. With every change that is made to the UI, there is a high risk that now there is more (or less) data required than before. Consequently, the backend needs to be adjusted as well to account for the new data needs. This kills productivity and notably slows down the ability to incorporate user feedback into a product.

With GraphQL, this problem is solved. Thanks to the flexible nature of GraphQL, changes on the client-side can be made without any extra work on the server. Since clients can specify their exact data requirements, no backend engineer needs to make adjustments when the design and data needs on the frontend change.

Benefits of a Schema & Type System

GraphQL uses a strong type system to define the capabilities of an API. All the types that are exposed in an API are written down in a schema using the GraphQL Schema Definition Language (SDL). This schema serves as the contract between the client and the server to define how a client can access the data.

Once the schema is defined, the teams working on frontend and backends can do their work without further communication since they both are aware of the definite structure of the data that’s sent over the network.

Frontend teams can easily test their applications by mocking the required data structures. Once the server is ready, the switch can be flipped for the client apps to load the data from the actual API.

Declarative Data Fetching

GraphQL embraces declarative data fetching with its queries.

The client selects data along with its entities with fields across relationships in one query request.

GraphQL decides which fields are needed for its UI, and it almost acts as UI-driven data fetching.

To retrieve all data in one request, a GraphQL query selects only the part of the data for the UI makes perfect sense.

It offers a great separation of concerns: a client knows about the data requirements; the server knows about the data structure and how to resolve the data from a data source (e.g. database, microservice, third-party API).

GraphQL Schema Stitching

Schema stitching makes it possible to create one schema out of multiple schemas.

Think about a microservices architecture for your backend where each microservice handles the business logic and data for a specific domain.

In this case, each microservice can define its own GraphQL schema, after which you'd use schema stitching to weave them into one that is accessed by the client.

Each microservice can have its own GraphQL endpoint, where one GraphQL API gateway consolidates all schemas into one global schema.

GraphQL Introspection

A GraphQL introspection makes it possible to retrieve the GraphQL schema from a GraphQL API.

Since the schema has all the information about data available through the GraphQL API, it is perfect for autogenerating API documentation.

It can also be used to mock the GraphQL schema client-side, for testing or retrieving schemas from multiple microservices during schema stitching.

GraphQL Versioning

In GraphQL there are no API versions as there used to be in REST.

In REST it is normal to offer multiple versions of an API (e.g. api.domain.com/v1/, api.domain.com/v2/), because the resources or the structure of the resources may change over time.

In GraphQL it is possible to deprecate the API on a field level.

Thus a client receives a deprecation warning when querying a deprecated field.

After a while, the deprecated field may be removed from the schema when not many clients are using it anymore.

This makes it possible to evolve a GraphQL API over time without the need for versioning.

Disadvantages:

GraphQL Query Complexity

People often mistake GraphQL as a replacement for server-side databases, but it’s just a query language.

Once a query needs to be resolved with data on the server, a GraphQL agnostic implementation usually performs database access.

GraphQL isn’t opinionated about that. Also, GraphQL doesn’t take away performance bottlenecks when you have to access multiple fields (authors, articles, comments) in one query.

Whether the request was made in a RESTful architecture or GraphQL, the varied resources and fields still have to be retrieved from a data source.

As a result, problems arise when a client requests too many nested fields at once.

Frontend developers are not always aware of the work a server-side application has to perform to retrieve data, so there must be a mechanism like maximum query depths, query complexity weighting, avoiding recursion, or persistent queries for stopping inefficient requests from the other side.

GraphQL Rate Limiting

Another problem is rate limiting.

Whereas in REST it is simpler to say “we allow only so many resource requests in one day”, it becomes difficult to make such a statement for individual GraphQL operations, because it can be everything between a cheap or expensive operation.

That’s where companies with public GraphQL APIs come up with their specific rate limiting calculations which often boil down to the previously mentioned maximum query depths and query complexity weighting.

GraphQL Caching

Implementing a simplified cache with GraphQL is more complex than implementing it in REST.

In REST, resources are accessed with URLs, so you can cache on a resource level because you have the resource URL as identifier.

In GraphQL, this becomes complex because each query can be different, even though it operates on the same entity.

You may only request just the name of an author in one query, but want to know the email address in the next.

That’s where you need a more fine-grained cache at field level, which can be difficult to implement.

However, most of the libraries built on top of GraphQL offer caching mechanisms out of the box.

GraphQL implementations

Server:

- C# / .NET
- Clojure
- Elixir
- Erlang
- Go
- Groovy
- Java
- JavaScript
- PHP
- Python
- Scala
- Ruby

Client:

- C# / .NET
- Clojurescript
- Go
- Java / Android
- JavaScript
- Swift / Objective-C iOS
- Python

- A GraphQL operation is either a query (read), mutation (write), or subscription (continuous read).

  Each of those operations is only a string that needs to be constructed according to the GraphQL query language specification.

- Once this GraphQL operation reaches the backend application, it can be interpreted against the entire GraphQL schema there, and resolved with data for the frontend application.

- GraphQL is not opinionated about the network layer, which is often HTTP, nor about the payload format, which is usually JSON.

  It isn’t opinionated about the application architecture at all.

  It is only a query language.

  GraphQL in a nutshell: The server application offers a GraphQL schema, where it defines all available data with its hierarchy and types, and a client application only queries the required data.

References:

- 4 years of GraphQL
- React.js Conf 2015 - Data fetching for React applications at Facebook
- Graphql Introduction
- Intro to GraphQL
- Specefication
- How to GraphQL
- Exploring GraphQL video
- Your first GraphQL server
- Zero to GraphQL in 30 minutes
- From Rest to GraphQL