GraphQL is the better REST

Over the past decade, REST has become the standard (yet a fuzzy one) for designing web APIs. It offers some great ideas, such as *stateless servers* and *structured access to resources*. However, REST APIs have shown to be too inflexible to keep up with the rapidly changing requirements of the clients that access them.

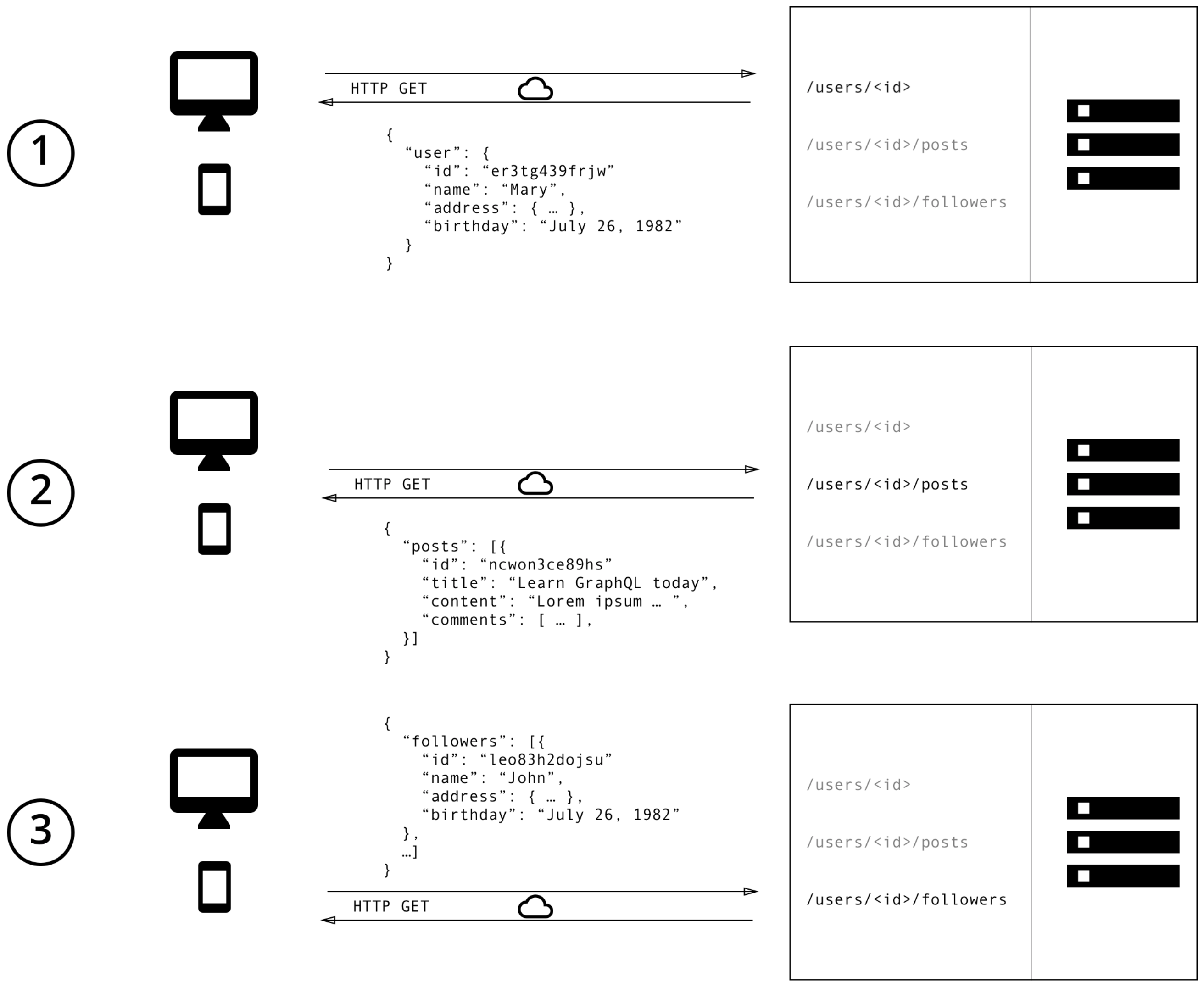
GraphQL was developed to cope with the need for more flexibility and efficiency! It solves many of the shortcomings and inefficiencies that developers experience when interacting with REST APIs.

To illustrate the major differences between REST and GraphQL when it comes to fetching data from an API, let's consider a simple example scenario: In a blogging application, an app needs to display the titles of the posts of a specific user. The same screen also displays the names of the last 3 followers of that user. How would that situation be solved with REST and GraphQL?

> 💡 Check out this article to learn more about why developers love GraphQL.

Data Fetching with REST vs GraphQL

With a REST API, you would typically gather the data by accessing multiple endpoints. In the example, these could be `/users/<id>` endpoint to fetch the initial user data. Secondly, there's likely to be a `/users/<id>/posts` endpoint that returns all the posts for a user. The third endpoint will then be the `/users/<id>/followers` that returns a list of followers per user.



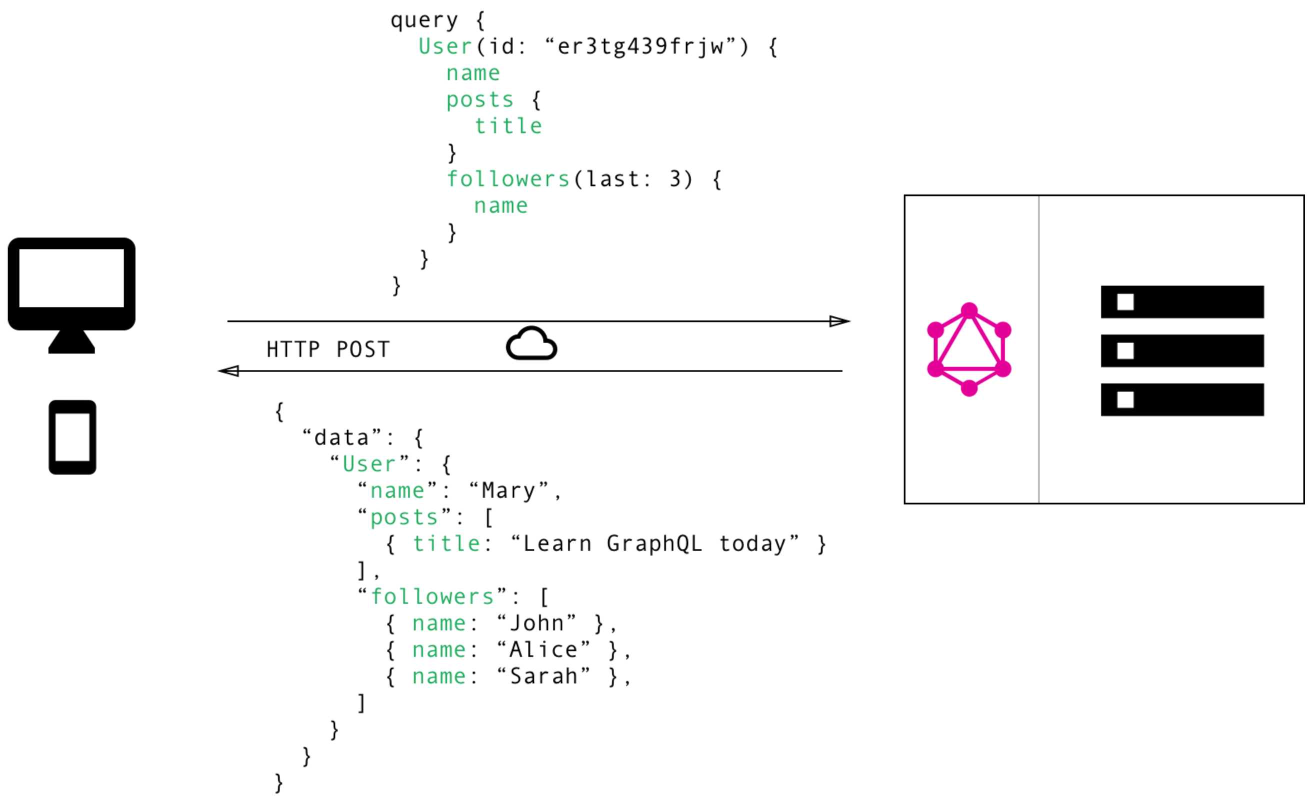
```
{
  “user”: {
    “id”: “er3tg439frjw”
    “name”: “Mary”,
    “address”: { … },
    “birthday”: “July 26, 1982”
  }
}
```

```
{
  “posts”: [{
    “id”: “ncwon3ce89hs”
    “title”: “Learn GraphQL today”,
    “content”: “Lorem ipsum … ”,
    “comments”: [ … ],
  }]
}
```

```
{
  “followers”: [{
    “id”: “leo83h2dojsu”
    “name”: “John”,
    “address”: { … },
    “birthday”: “July 26, 1982”
  },
  …]
}
```

With REST, you have to make three requests to different endpoints to fetch the required data. You’re also *overfetching* since the endpoints return additional information that’s not needed.

In GraphQL on the other hand, you’d simply send a single query to the GraphQL server that includes the concrete data requirements. The server then responds with a JSON object where these requirements are fulfilled.



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

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

Using GraphQL, the client can specify exactly the data it needs in a *query*. Notice that the *structure* of the server’s response follows precisely the nested structure defined in the query.

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.

> “Think in graphs, not endpoints.” Lessons From 4 Years of GraphQL by Lee Byron, GraphQL Co-Inventor.

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.

As an example, consider the same app would also need to display the last three followers per user. The API provides the additional endpoint `/users/<user-id>/followers`. In order to be able to display the required information, the app will have to make one request to the `/users` endpoint and then hit the `/users/<user-id>/followers` endpoint for *each* user.

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.

Insightful Analytics on the Backend

GraphQL allows you to have fine-grained insights about the data that's requested on the backend. As each client specifies exactly what information it's interested in, it is possible to gain a deep understanding of how the available data is being used. This can for example help in evolving an API and deprecating specific fields that are not requested by any clients any more.

With GraphQL, you can also do low-level performance monitoring of the requests that are processed by your server. GraphQL uses the concept of *resolver functions* to collect the data that's requested by a client. Instrumenting and measuring performance of these resolvers provides crucial insights about bottlenecks in your system.

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.