Intro to GraphQL

There's a problem with using traditional REST APIs: we always get the same server response for a given endpoint.

This lack of flexibility can lead to two equally annoying issues:

1. *Over-fetching* data — when we get a huge response but only need a small piece of it.
2. *Under-fetching* — when we perform the first call only to receive a small piece of what we need. This can lead to chaining API calls and fetching even more unnecessary data in an attempt to get what we need.

Wouldn't it be great instead if we could shape the response we get with the request we make, and fetch only what we need, when we need it? Fortunately that is possible with GraphQL, a modern way to build and query APIs.

In Vue Mastery's GraphQL course, we will be creating a Vue + GraphQL app together to gain the confidence of using this powerful, modern tool.

Understanding its unique benefits

In a nutshell, GraphQL is a syntax that describes how to ask for data, and it's generally used to load data from the server.

One of the core features and the biggest benefits of GraphQL is that *you get only the data you asked for.* Unlike with a REST API, where the shape of the API call response is defined on the server, with GraphQL this is done on the client side. This allows us to make a single request to fetch all required information and nothing else, instead of making subsequent REST API calls.

To understand this difference better, let's look at an example where we'll compare fetching data from GitHub using both REST and GraphQL APIs.

Fetching with REST

Imagine that we want to fetch a list of repositories of a certain user and check what programming languages are used in those repos.

With a REST API, first we would perform a call for `/users/:username/repos`, which would get us a list of repositories:

```
// fetch with ==> /users/NataliaTepluhina/repos
// and get this response:

[
  {
    "id": 328931302,
    "name": "api-wheel-demo",
    "full_name": "NataliaTepluhina/api-wheel-demo",
    "languages_url": "https://api.github.com/repos/NataliaTepluhina/api-
wheel-demo/languages",,
    ...
  },
  {
    "id": 243311967,
    "name": "apollo-presentation",
    "full_name": "NataliaTepluhina/apollo-presentation",
    "languages_url": "https://api.github.com/repos/NataliaTepluhina/apollo-
presentation/languages",
    ...
  }
]
```

(Please note that I'm not showing the entire repo object since they are huge objects with more than 60 fields, some of them nested.)

But what if we wanted to know the programming languages used in each repo? Looking at the data, we only have the `languages_url` links to retrieve languages with. This means we need to take another step and iterate over the repositories list, fetching languages for each one, using that `languages_url`.

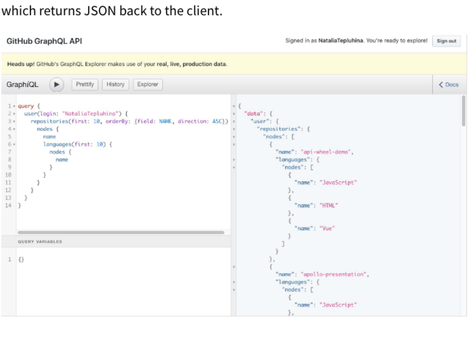
```
// fetch with ==> /repos/NataliaTepluhina/api-wheel-demo/languages
// and get this response:

{
  "Vue": 1339,
  "JavaScript": 1167,
  "HTML": 611
}

// fetch with ==> /repos/NataliaTepluhina/apollo-presentation/languages
// and get this response:
{
  "Vue": 3925,
  "JavaScript": 2887,
  "HTML": 605
}
```

Fetching with GraphQL

So how is this different with GraphQL? Using it, we send a query string that specifies the fields we want to fetch. This string is interpreted on the server, which returns JSON back to the client.



(What we're looking at in the image is `GraphiQL` — an explorer that lets you examine the GraphQL API.)

As you can see, with only one request we fetched everything we needed without any additional, over-fetched data!

An added benefit of GraphQL is that it uses a strong type system to describe all of the possible data that you can fetch from the GraphQL server. This set of types is known as a *GraphQL schema* and can be examined in a UI like GraphiQL. Strong types make GraphQL less error-prone, can be validated during compile time, and can use IDE tools for validation and autocompletion.

What to expect from the course

Now that we are starting to understand how GraphQL provides a unique ability to query our data, how do we work with it on the frontend? We could use POST requests, but it's more convenient to use one of the GraphQL clients.

In this course, we will be focusing on Apollo Client and its Vue integration: VueApollo. We'll learn how to fetch and change data on the server, how to work with real-time subscriptions, and even learn how you can replace your local state management tool like Vuex with a built-in Apollo Client cache.

To get the most out of this course, you'll need a solid understanding of Vue itself and the Vue CLI, as well as a fundamental understanding of the Composition API. So if you're ready to start feeling confident plugging GraphQL into your Vue apps, I'll see you in the next lesson.

Setting up local state with Apollo Client

Unlock this lesson by subscribing to a plan.

Unlock Content

Fetching data with queries

In this lesson, we will learn how to make simple queries to fetch data from the GraphQL API and how to work with this data in Vue components.

Improving Developer Experience

Unlock this lesson by subscribing to a plan.

Unlock Content

Query variables, handling loading and errors

Advanced queries

In the previous lesson, we created a typeahead search by passing a parameter to

Unlock this lesson by subscribing to a plan.

Unlock Content

Updating data with mutations

S
A
G

Unlock this lesson by subscribing to a plan.

Unlock Content

Manual cache updates and optimistic responses

Unlock this lesson by subscribing to a plan.

Unlock Content

Real-time updates with subscriptions

Unlock this lesson by subscribing to a plan.

Unlock Content

Bonus: Q&A with Natalia Tepluhina

Unlock this lesson by subscribing to a plan.

Unlock Content