GraphQL: The Great Aggregator

Paweł Fielek Follow
Dec 4, 2018 · 6 min read

Many backend applications are tailored to a small handful of client applications' specific needs. This makes a lot of sense, especially with the massive influx of fullstack engineering teams we've seen within the past decade. Here at VICE, we treat our API ecosystem a little bit differently. Our APIs are their own product with their own dedicated product manager and engineering team. This methodology helps us write

APIs that can be consumed by a wider array of internal and third party client applications.

This is by no means a new approach to architecting an organization's applications and it comes with a lot of benefits. It's great for scalability and allows a relatively small team to maintain a pretty large breadth of endpoints. It also allows us to focus on building services with more individualized purposes rather than monoliths that quickly become difficult to maintain.

There are some drawbacks though. One of the biggest issues is that in order to satisfy all clients calling an endpoint, we must serve a lot of data. Let's say we have two client applications, a web client and a mobile app client, consuming the `GET /articles` endpoint. The web client only needs to know about an article's title, topic, and description. Meanwhile, the mobile app client wants the article's title and author. In order to support both of these requirements, our payload for each individual article needs to look something like this:

```
{
  "title": "Knickers the Massive Cow Is Here to Make 2018 Worth It",
  "topics": [
    {
      "name": "animals",
      "url": "https://www.vice.com/en_us/topic/animals"
    },
    {
      "name": "cows",
      "url": "https://www.vice.com/en_us/topic/cows"
    }
  ],
  "description": "He's extremely large and everyone loves him.",
  "author": "Nicole Clark"
}
```

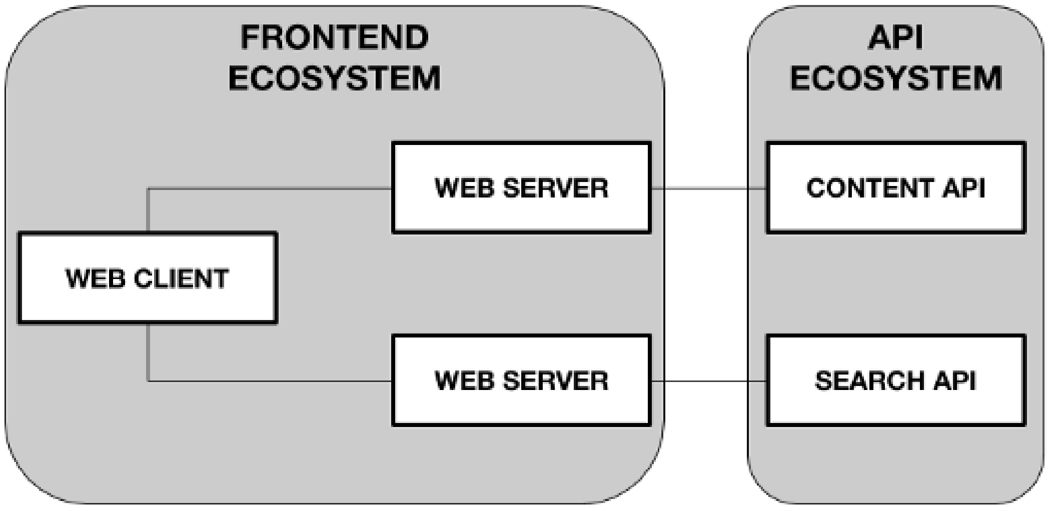
Both clients now have the data they need, as well as some extra data that they will simply ignore. While this example payload is pretty small, a typical VICE article object can be as small as 6 or 7KB and as large as 100KB+. The vice.com homepage has 18 articles displayed on it at the time of this writing, which means that we can end up serving over 1MB of data just so the web application can display a collection of links and thumbnail images (note: the 1MB payload does not include the additional image files, fonts, etc. that need to be downloaded by the client).

Another issue that we sometimes run into is that there's some confusion among client application developers as to which endpoints they should be making requests to in order to fulfill requests or perform actions. Scouring documentation that is sometimes not up-to-date or not in a centralized place can hinder work on new features.

Enter GraphQL

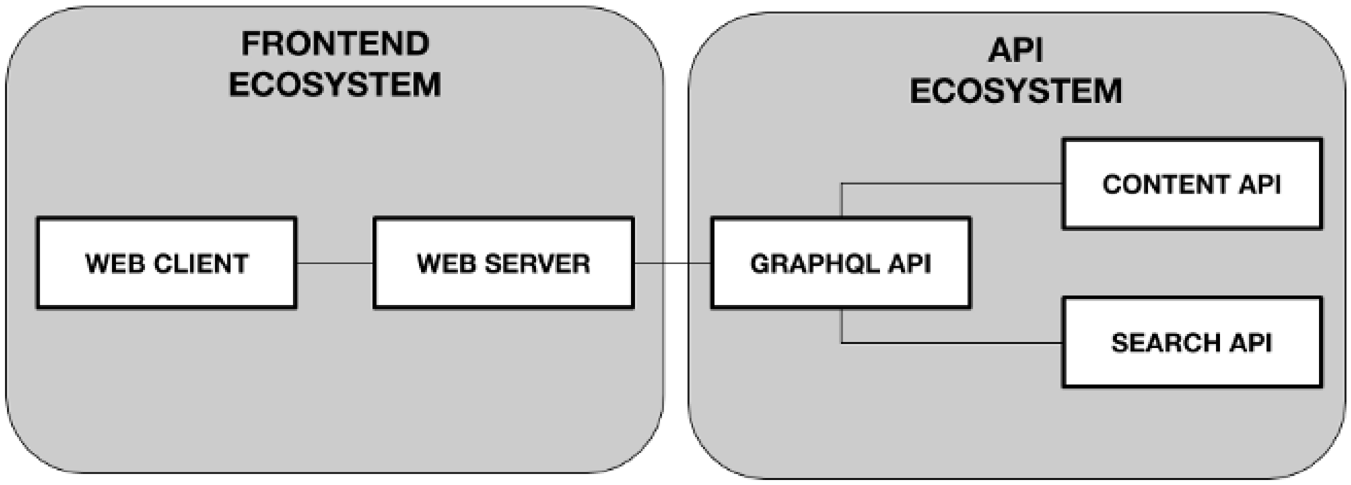
GraphQL solves both of these problems for us in one go! We can start by going over how we had organized our services in our already existing API ecosystem.

In this scenario, our client web application needs to render an article and queue up a second article for infinite scroll. We'd like to make sure the second article relates to the first article, so we're going to use the search service in order to find a related article to display as the user scrolls down the page (you can read more on how we leveraged elasticsearch in order to drive better story discovery here). One can imagine how many more services the client needs to know about as the application expands and becomes more complex.



Example of how article page data was requested under the original setup.

The GraphQL approach allows us to aggregate all of our service endpoints into one large schema, which is centralized, so that developers working on client applications do not need to go digging for the correct set of documents in order to learn about the format for a response. Our APIs ecosystem becomes a little more complex with the additional layer of abstraction, but becomes a lot simpler for the developers working on client applications.



Example of how article page data is requested under the GraphQL setup

An additional benefit is that now the client can fetch all of its data in a single, simple query, since the client's entrypoint into our APIs is now centralized.

```
{
  articles(article_id: "123abc"){
    title
    description
    body
    thumbnail_url
    author{
      full_name
    }
  }
  related(article_id: "123abc") {
    title
    description
    body
    thumbnail_url
    author{
      full_name
    }
  }
}
```

Reducing data

It’s important to note that unless you are writing your database queries to return fewer fields of data, you will need to perform data reductions at some point or another if you wish to decrease the size of a payload. The services-oriented approach to doing this would be to create an entirely new service just for this purpose, which is exactly what we did.

Now let’s take another look at our example of the vice.com homepage payload size. Since VICE is a media company and most users consume media on their phones nowadays, the biggest performance bottleneck is the user’s network speed. In order to keep the payload size we deliver to the user’s web client manageable, the web application’s server would have to call the content API for all data, then reduce that data down to just the fields that it needs in order to render the homepage view, and finally deliver that pre-rendered view to the user’s device. While filtering out unneeded data was an effective stopgap measure for the web application, it seems a little redundant for client applications to always have to fetch more data than they need and reduce that data themselves.

GraphQL's purpose is to mutate and reduce data to a client's whim. Now, if the web client needs only article titles, thumbnails, urls, descriptions, and topics, it can request just that data and nothing more.

```
{
  articles(locale: "en_us"){
    title
    thumbnail_url
    url
    description
    topics{
      name
      url
    }
  }
}
```

Within the confines of a single service, the database tends to be the largest performance bottleneck. In order to combat this, we keep a Redis cache for recently run database queries. While it's possible to allow clients to structure their own database calls to only return necessary fields, we'd run into the issue of having to create multiple results caches for items in their various queried formats. This method would also mean that more cached content would have to be invalidated when an article is updated. This would lead to a far greater strain on our database, given that every client application would need differently structured data. We're able to avoid these caching issues by not allowing clients to define their desired database fields.

The savings

How much does GraphQL save us on payload size? Overall we have found an average reduction in payload size of about 90%. The time for the API request via the GraphQL service rather than going directly to the content API to complete is approximately 50% longer, however, the client ends up with a faster experience thanks to our redux store (along with the now smaller GraphQL payload) being saved within our CDN cache.

Looking towards the future

Next steps for our GraphQL service will be to allow for more dynamic schema generation. In its current state, schemas must be written out explicitly for the service.

Moving forward, we'd like to add the ability to convert JSON API schemas to GraphQL schemas for the use of future services which will adhere to the JSON API specifications.

GraphQL Nodejs API Backend Microservices