GraphQL Service Layer to Enable Client-driven, Optimized and Secure Front-end Architecture

Amit Jambusaria[1]

[1]*Amit Jambusaria; Syracuse University; 31 Caspar St, Boston, MA - 02132*

---***---

Abstract - *GraphQL is an open-source data query and modification language for APIs leveraged by several prominent tech shops such as Facebook (original creator), Github, Pinterest, Intuit, Cousera, Paypal, Yelp and Shopify to name a few. It has gotten a lot of positive attention from the engineering community and supporters have termed it as "Better REST", asserting a range of benefits over traditional REST. With all the buzz around GraphQL, do developers need to make an active shift towards it and deprecate REST endpoints completely? It depends on many factors, which are discussed in this article. As part of the study, we will be also be establishing performance benchmarks of GraphQL vs REST in terms of overall web page load time (TTI)..*

Key Words: **GraphQL; Web Applications; Front-end; API; REST; Services; Client architecture; Data Query**

1.INTRODUCTION

GraphQL is a data-query language created by Facebook that went open source in 2015. It provides a complete understandable description of the data in the API and enables clients to function in a declarative style to fetch exactly what they need - nothing less, nothing more. In this article, we will summarize how it works, comparative study of GraphQL vs REST, why you should use it and what are some of its drawbacks. Graph data structures are connection maps, and they have one key advantage over both relational / document stores - you can express both relational & hierarchical information as graphs.

RESTful APIs follow clear and well-structured resource-oriented approach, when the data gets more complex, the routes get longer. It is not always possible to fetch data with a single request. GraphQL structures data in the form of a graph with its powerful query syntax to request, retrieve and update data. GraphQL is a query language for APIs and not for databases.

2. GRAPHQL COMPONENTS

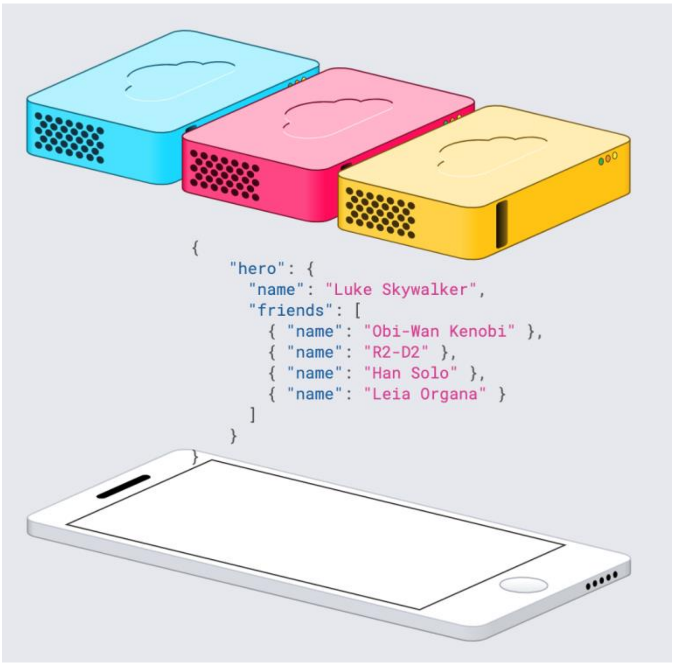
GraphQL enables you to fetch (or modify) all the data on a server in one go. Your client can make HTTP requests to the /graphql endpoint by providing query, variables and operationName. For example, the request would look something like –

```
{
        “operationName”: “operation name”,
        “query”: “query string”,
        “variables”: “variables”
}
```

GraphQL allows for three kinds of operations -

1. *Query*
2. *Mutation*
3. *Subscription*

A GraphQL operation is either a query (read), mutation (write), or subscription (continuous read). Each of these operations is only a string that needs to be constructed according to the GraphQL specification. Once this operation reaches the backend application, it can be interpreted against the entire GraphQL schema there and resolved with data for the frontend application.



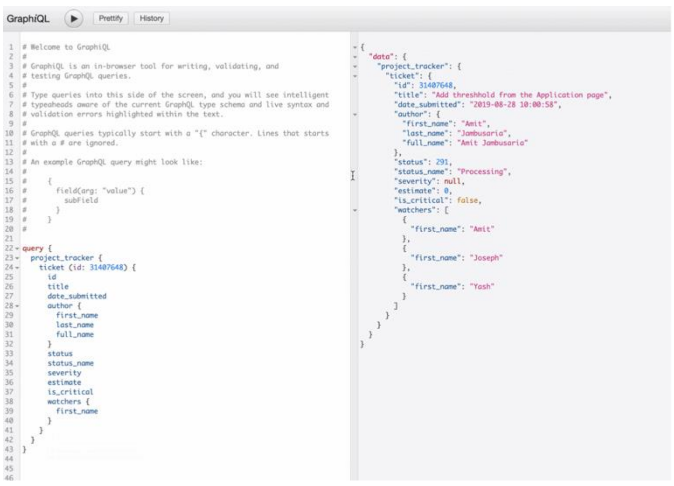
```
{
  "hero": {
    "name": "Luke Skywalker",
    "friends": [
      { "name": "Obi-Wan Kenobi" },
      { "name": "R2-D2" },
      { "name": "Han Solo" },
      { "name": "Leia Organa" }
    ]
  }
}
```

2.1 QUERY

Query enables the client to fetch data from the server. They are comparable to the GET calls in traditional REST. A GraphQL query is a string that is sent to a server to be interpreted and fulfilled, which then returns JSON back to the client. At a very basic level, each query contains fields and arguments.

The shape of the query is of the same shape as the result. This is a key feature in GraphQL, because you always get back what you expect, and the server knows exactly what fields the client is asking for. Let's take the example of the 'Ticket' resource given below. The query call is made from a web IDE called Graphiql that helps you to test and structure your queries against your server. We have defined a query that pulls ticket fields by passing in the ticket ID as an identifier argument.

GraphQL queries access not just the properties of one resource but also smoothly follow references between them. While typical REST APIs require loading from multiple URLs, GraphQL APIs get all the data your app needs in a single request. Apps using GraphQL can be quick even on slow mobile network connections. Existing APIs fetch more from the GET calls than what is required, leading to the emergence of technologies like GraphQL instead of traditional ones like REST [1].



```
# Welcome to GraphiQL
#
# GraphiQL is an in-browser tool for writing, validating, and
# testing GraphQL queries.
#
# Type queries into this side of the screen, and you will see intelligent
# typeaheads aware of the current GraphQL type schema and live syntax and
# validation errors highlighted within the text.
#
# GraphQL queries typically start with a "{" character. Lines that starts
# with a # are ignored.
#
# An example GraphQL query might look like:
#
#     {
#       field(arg: "value") {
#         subField
#       }
#     }
#

query {
  project_tracker {
    ticket (id: 31407648) {
      id
      title
      date_submitted
      author {
        first_name
        last_name
        full_name
      }
      status
      status_name
      severity
      estimate
      is_critical
      watchers {
        first_name
      }
    }
  }
}
```

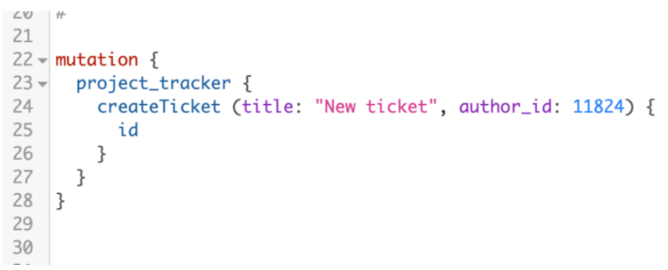
```
{
  "data": {
    "project_tracker": {
      "ticket": {
        "id": 31407648,
        "title": "Add threshhold from the Application page",
        "date_submitted": "2019-08-28 10:00:58",
        "author": {
          "first_name": "Amit",
          "last_name": "Jambusaria",
          "full_name": "Amit Jambusaria"
        },
        "status": 291,
        "status_name": "Processing",
        "severity": null,
        "estimate": 0,
        "is_critical": false,
        "watchers": [
          {
            "first_name": "Amit"
          },
          {
            "first_name": "Joseph"
          },
          {
            "first_name": "Yash"
          }
        ]
      }
    }
  }
}
```

Fig - 2: Graphiql interface to test GraphQL queries

Queries can also parse through dynamic arguments which can be passed in JSON format as variables. In the example above, instead of hardcoding the ticket ID, you could pass in a variable $id. Given below is the query network call as seen in the browser dev tools.

2.1 MUTATION

Just reading data from the server is not enough. We also need the ability to create, update and delete data from the server. In REST, this is accomplished by POST with a payload (usually JSON) that is passed to the server. In GraphQL this is solved through Mutation. Mutations enable the client to modify data on the server-side. In our example around "Ticket", we will replace the query with the mutation keyword. The corresponding mutation type also needs to exist on the server-side. This is an example:



```
#

mutation {
  project_tracker {
    createTicket (title: "New ticket", author_id: 11824) {
      id
    }
  }
}
```

Fig - 3: An example of Mutation to create / update data

In our example, the createTicket mutation accepts two arguments for creating a ticket - title and author_id. On ticket creation, we return the $id of the newly created ticket. Just like the Query, Mutation is a root object type. Mutations for the most part are very flexible and can return whatever you desire: scalars such as int, string, bool and core types like the Ticket, or even custom response objects. Similar to queries, if the mutation field returns an object type, you can ask for nested fields.

2.3 SUBSCRIPTION

Subscription enables the client to fetch real-time updates from the server. You can think of subscriptions as analogous to continuous polling mechanisms. It makes it possible for the server to stream data to all the clients that are listening or "subscribed" to it. Just like queries, subscriptions allow you to read data. *Unlike* queries, subscriptions maintain an active connection to your GraphQL server, most commonly via WebSocket. This enables your server to push updates to the client over time. Executing a subscription creates a persistent function on the server that maps an underlying Source Stream to a returned Response Stream. You can define available subscriptions in your GraphQL schema as fields of the Subscription type.

So, when should you use subscriptions? In most use cases, you should not require subscriptions. Your client can stay consistent with the backend by querying intermittently or re-execute queries on demand based on triggers / actions from the user. Subscriptions are a great for

1. **Small incremental changes to large objects -** If your backend object is continuously being updated, re-querying constantly from the client can get expensive and slow. For example, consider the stock trading apps - Robinhood, WeBull, Fidelity etc. The ticker price for a given stock is constantly fluctuating. Rather than persistently querying the server, subscriptions would provide a much more elegant solution. You can fetch the initial state of the object with a query, and your server can proactively push updates to individual fields as they occur.

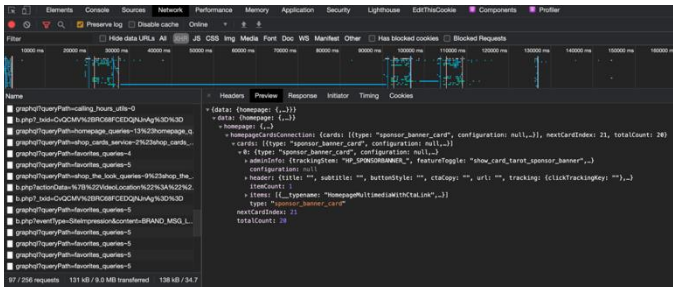
2. **Low-latency, real-time updates** - A chat application is a good example where all the chat participants need to receive messages as soon as they are posted.

3. PAGE LOAD TIME OPTIMIZATION STUDY

As part of web performance measurement, following key metrics needs to be considered –

- **First contentful paint (FCP):** measures the time from when the page starts loading to when any part of the page's content is rendered on the screen. (lab, field)
- **Largest contentful paint (LCP):** measures the time from when the page starts loading to when the largest text block or image element is rendered on the screen. (lab, field)
- **First input delay (FID):** measures the time from when a user first interacts with your site (i.e. when they click a link, tap a button, or use a custom, JavaScript-powered control) to the time when the browser is actually able to respond to that interaction. (field)
- **Time to Interactive (TTI):** measures the time from when the page starts loading to when it's visually rendered, its initial scripts (if any) have loaded, and it's capable of reliably responding to user input quickly. (lab)
- **Total blocking time (TBT):** measures the total amount of time between FCP and TTI where the main thread was blocked for long enough to prevent input responsiveness. (lab)
- **Cumulative layout shift (CLS):** measures the cumulative score of all unexpected layout shifts that occur between when the page starts loading and when its lifecycle state changes to hidden. (lab, field)

We will be measuring the Time to Interactive (TTI) for web pages of different sizes using regular REST calls vs GraphQL service layer. In the research the web page content is unaltered. Fig - 4 details the GraphQL network calls and the server response time.

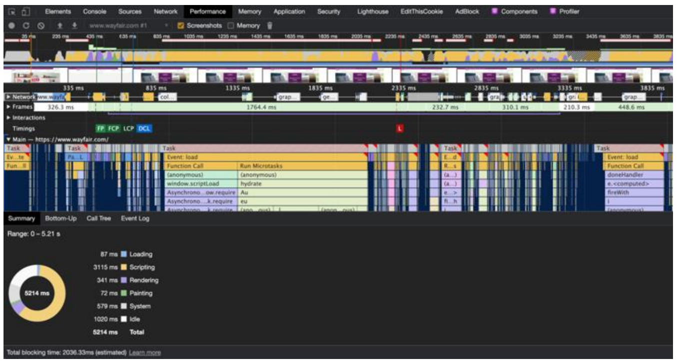


```
Name
graphql?queryPath=calling_hours_utils~0
b.php?_txid=CvQCMV%2BRC68FCEDQjNJnAg%3D%3D
graphql?queryPath=homepage_queries~13%23homepage_q...
graphql?queryPath=shop_cards_service~2%23shop_cards_...
graphql?queryPath=favorites_queries~4
graphql?queryPath=favorites_queries~5
graphql?queryPath=shop_the_look_queries~9%23shop_the_...
b.php?actionData=%7B%22VideoLocation%22%3A%22%2...
b.php?_txid=CvQCMV%2BRC68FCEDQjNJnAg%3D%3D
graphql?queryPath=favorites_queries~5
b.php?eventType=SiteImpression&content=BRAND_MSG_L...
graphql?queryPath=favorites_queries~5
graphql?queryPath=favorites_queries~5
graphql?queryPath=favorites_queries~5
graphql?queryPath=favorites_queries~5
97 / 256 requests   131 kB / 9.0 MB transferred   138 kB / 34.7
```

```
▾{data: {homepage: {,…}}}
  ▾data: {homepage: {,…}}
    ▾homepage: {,…}
      ▾homepageCardsConnection: {cards: [{type: "sponsor_banner_card", configuration: null,…}], nextCardIndex: 21, totalCount: 20}
        ▾cards: [{type: "sponsor_banner_card", configuration: null,…}]
          ▾0: {type: "sponsor_banner_card", configuration: null,…}
            ▸adminInfo: {trackingStem: "HP_SPONSORBANNER_", featureToggle: "show_card_tarot_sponsor_banner",…}
             configuration: null
            ▸header: {title: "", subtitle: "", buttonStyle: "", ctaCopy: "", url: "", tracking: {clickTrackingKey: ""},…}
             itemCount: 1
            ▸items: [{__typename: "HomepageMultimediaWithCtaLink",…}]
             type: "sponsor_banner_card"
         nextCardIndex: 21
         totalCount: 20
```

Fig - 4: GraphQL network calls to the server

In Fig – 5, we see performance profiler graphical representation of the execution time, most of the time is consumed during scripting and then for rendering.

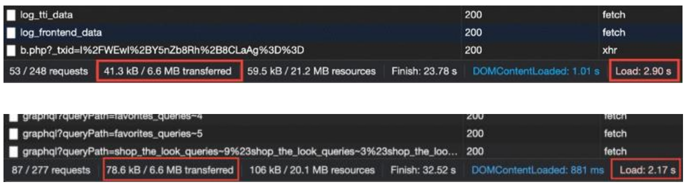


```
Range: 0 – 5.21 s

  87 ms  Loading
3115 ms  Scripting
 341 ms  Rendering
  72 ms  Painting
 579 ms  System
1020 ms  Idle
5214 ms  Total

Total blocking time: 2036.33ms (estimated) Learn more
```

Fig - 5: Browser Performance Profiler

Measuring the GraphQL performance in terms of Page load times for web pages of different sizes.

Case 1:
Web page size – 6.6MB
REST page load time – 2.90 sec
GraphQL page load time – 2.17 sec
Delta / Performance optimization – 0.73 sec

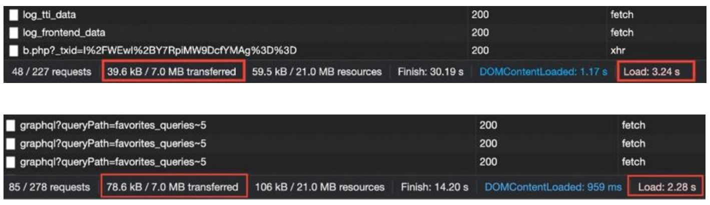


```
log_tti_data                                               200     fetch
log_frontend_data                                          200     fetch
b.php?_txid=l%2FWEwl%2BY5nZb8Rh%2B8CLaAg%3D%3D             200     xhr
53 / 248 requests | 41.3 kB / 6.6 MB transferred | 59.5 kB / 21.2 MB resources | Finish: 23.78 s | DOMContentLoaded: 1.01 s | Load: 2.90 s
```

```
graphql?queryPath=favorites_queries~4                      200     fetch
graphql?queryPath=favorites_queries~5                      200     fetch
graphql?queryPath=shop_the_look_queries~9%23shop_the_look_queries~3%23shop_the_loo...  200  fetch
87 / 277 requests | 78.6 kB / 6.6 MB transferred | 106 kB / 20.1 MB resources | Finish: 32.52 s | DOMContentLoaded: 881 ms | Load: 2.17 s
```

Fig – 6: Page load time – REST vs GQL (Case 1)

Case 2:
Web page size – 7MB
REST page load time – 3.24 sec
GraphQL page load time – 2.28 sec
Delta / Performance optimization – 0.96 sec

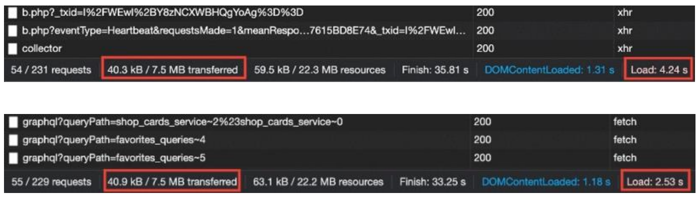


```
log_tti_data                                               200     fetch
log_frontend_data                                          200     fetch
b.php?_txid=l%2FWEwl%2BY7RpiMW9DcfYMAg%3D%3D               200     xhr
48 / 227 requests | 39.6 kB / 7.0 MB transferred | 59.5 kB / 21.0 MB resources | Finish: 30.19 s | DOMContentLoaded: 1.17 s | Load: 3.24 s
```

```
graphql?queryPath=favorites_queries~5                      200     fetch
graphql?queryPath=favorites_queries~5                      200     fetch
graphql?queryPath=favorites_queries~5                      200     fetch
85 / 278 requests | 78.6 kB / 7.0 MB transferred | 106 kB / 21.0 MB resources | Finish: 14.20 s | DOMContentLoaded: 959 ms | Load: 2.28 s
```

Fig – 7: Page load time – REST vs GQL (Case 2)

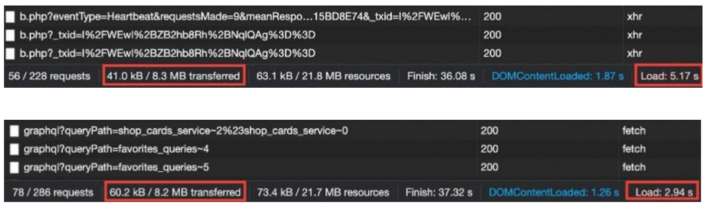
Case 3:
Web page size – 7.5MB
REST page load time – 4.24 sec
GraphQL page load time – 2.53 sec
Delta / Performance optimization – 1.71 sec



```
b.php?_txid=I%2FWEwI%2BY8zNCXWBHQgYoAg%3D%3D                                   200   xhr
b.php?eventType=Heartbeat&requestsMade=1&meanRespo...7615BD8E74&_txid=I%2FWEwI...  200   xhr
collector                                                                       200   xhr
54 / 231 requests | 40.3 kB / 7.5 MB transferred | 59.5 kB / 22.3 MB resources | Finish: 35.81 s | DOMContentLoaded: 1.31 s | Load: 4.24 s

graphql?queryPath=shop_cards_service~2%23shop_cards_service~0   200   fetch
graphql?queryPath=favorites_queries~4                           200   fetch
graphql?queryPath=favorites_queries~5                           200   fetch
55 / 229 requests | 40.9 kB / 7.5 MB transferred | 63.1 kB / 22.2 MB resources | Finish: 33.25 s | DOMContentLoaded: 1.18 s | Load: 2.53 s
```

Fig – 8: Page load time – REST vs GQL (Case 3)

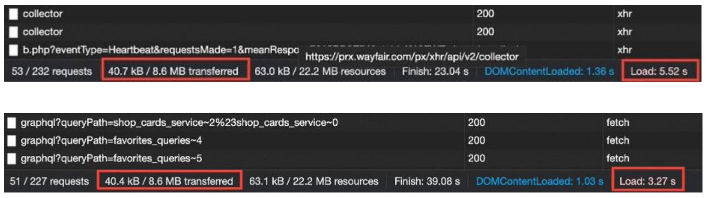
Case 4:
Web page size – 8.2MB
REST page load time – 5.17 sec
GraphQL page load time – 2.94 sec
Delta / Performance optimization – 2.23 sec



```
b.php?eventType=Heartbeat&requestsMade=9&meanRespo...15BD8E74&_txid=I%2FWEwI%...  200   xhr
b.php?_txid=I%2FWEwI%2BZB2hb8Rh%2BNqIQAg%3D%3D                                   200   xhr
b.php?_txid=I%2FWEwI%2BZB2hb8Rh%2BNqIQAg%3D%3D                                   200   xhr
56 / 228 requests | 41.0 kB / 8.3 MB transferred | 63.1 kB / 21.8 MB resources | Finish: 36.08 s | DOMContentLoaded: 1.87 s | Load: 5.17 s

graphql?queryPath=shop_cards_service~2%23shop_cards_service~0   200   fetch
graphql?queryPath=favorites_queries~4                           200   fetch
graphql?queryPath=favorites_queries~5                           200   fetch
78 / 286 requests | 60.2 kB / 8.2 MB transferred | 73.4 kB / 21.7 MB resources | Finish: 37.32 s | DOMContentLoaded: 1.26 s | Load: 2.94 s
```

Fig – 9: Page load time – REST vs GQL (Case 4)

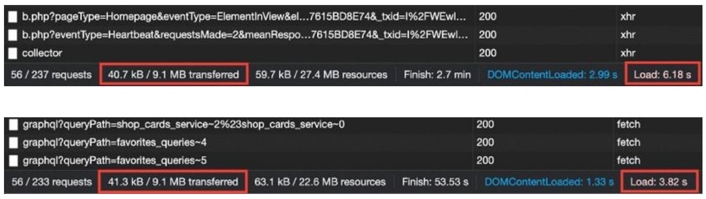
Case 5:
Web page size – 8.6MB
REST page load time – 5.52 sec
GraphQL page load time – 3.27 sec
Delta / Performance optimization – 2.25 sec



```
collector                                                  200   xhr
collector                                                  200   xhr
b.php?eventType=Heartbeat&requestsMade=1&meanRespo               xhr
                         https://prx.wayfair.com/px/xhr/api/v2/collector
53 / 232 requests | 40.7 kB / 8.6 MB transferred | 63.0 kB / 22.2 MB resources | Finish: 23.04 s | DOMContentLoaded: 1.36 s | Load: 5.52 s

graphql?queryPath=shop_cards_service~2%23shop_cards_service~0   200   fetch
graphql?queryPath=favorites_queries~4                           200   fetch
graphql?queryPath=favorites_queries~5                           200   fetch
51 / 227 requests | 40.4 kB / 8.6 MB transferred | 63.1 kB / 22.2 MB resources | Finish: 39.08 s | DOMContentLoaded: 1.03 s | Load: 3.27 s
```

Fig – 10: Page load time – REST vs GQL (Case 6)

Case 6:
Web page size – 9.1MB
REST page load time – 6.18 sec
GraphQL page load time – 3.82 sec
Delta / Performance optimization – 2.36 sec



```
b.php?pageType=Homepage&eventType=ElementInView&el...7615BD8E74&_txid=I%2FWEwI...  200   xhr
b.php?eventType=Heartbeat&requestsMade=2&meanRespo...7615BD8E74&_txid=I%2FWEwI...  200   xhr
collector                                                                        200   xhr
56 / 237 requests | 40.7 kB / 9.1 MB transferred | 59.7 kB / 27.4 MB resources | Finish: 2.7 min | DOMContentLoaded: 2.99 s | Load: 6.18 s

graphql?queryPath=shop_cards_service~2%23shop_cards_service~0   200   fetch
graphql?queryPath=favorites_queries~4                           200   fetch
graphql?queryPath=favorites_queries~5                           200   fetch
56 / 233 requests | 41.3 kB / 9.1 MB transferred | 63.1 kB / 22.6 MB resources | Finish: 53.53 s | DOMContentLoaded: 1.33 s | Load: 3.82 s
```

Fig – 11: Page load time – REST vs GQL (Case 6)

Final analysis (Chart 1)–

1. Comparing the page load time for REST vs GraphQL for web pages of different size, we find GraphQL is always the winner in terms of overall performance.
2. The delta / optimization value (in sec) increases i.e., becomes better with increase in the size of web page.

Web Page Size	Normal Page Load time	GraphQL Page Load Time
6.6mb	2.90 sec	2.17 sec
7mb	3.24 sec	2.28 sec
7.5mb	4.24 sec	2.53 sec
8.2mb	5.17 sec	2.94 sec
8.6mb	5.52 sec	3.27 sec
9.1mb	6.18 sec	3.82 sec

Chart - 1: Page load time value REST vs GraphQL

The style defines a set of constraints intended to improve performance, availability, and scalability and it is based on a traditional client-server paradigm [2].

4. GRAPHQL BENEFITS

GraphQL may be a good candidate for your API layer if your application has a data domain with highly interrelated, nested, or traversable concepts. These are the types of relationships that are difficult to model with a RESTful API and usually result in repeated round trips to the backend. Other reasons to consider GraphQL are: You want to give clients control over what data (fields) are returned, you want to reduce the amount of data sent per request (related to clients asking for what they want), leveraging information about what the client requests (query) allows you to more efficiently load and serve the data.

Following are the benefits of using GraphQL -

- Exact data fetching. GraphQL minimizes the data that needs to be transferred over the network. Your client can specify exactly what resource properties need to be pulled from the server, reducing the overall payload and complexity of the call.
- For example, in order to study the gains achieved by GraphQL due to the lack of over-fetching, Brito et al. [3] implemented 14 queries used in seven recent empirical software engineering papers.
- Nearly all externally facing REST APIs we know of trend or end up in these non-ideal states, as well as nearly all *internal* REST APIs. The consequences of opaqueness and over-fetching are more severe in internal APIs since their velocity of change and level of usage is almost always higher. [4]
- Single request can fetch multiple resources. Unlike traditional REST, a single GraphQL query call can fetch data across multiple backend resources. This reduces the over sprawl of

endpoints on the server which can be a big issue with REST.

- More power to the client. One of the key benefits of GraphQL is that the client has complete control over data fetching. Rather than the server responding with static payloads, the client can dynamically request necessary data points.
- Schema stitching. Great fit for complex systems with microservices. By integrating multiple systems behind its API, GraphQL unifies them and hides their complexity.
- Highly reusable code means that everyone benefits from everyone else's work. When types (schema objects) get added, everyone gets to use them without writing a single line of new code.
- Enables parallel processing. GraphQL allows for loading fields at the same level in parallel. For example, for a given user if you want to fetch medical history and employment history, you can load those in parallel.
- Discovery of types i.e. figuring out what other people have already made - is easy when using tools like Voyager, GraphQL Playground and Graphiql. This helps in reducing duplicate code and helps achieve the DRY principle.
- First class server-side rendering support lets you get stuff done faster. See Server-side rendering with GraphQL for detailed explanation and examples.
- Amazing tooling and community. GraphQL Playground and Graphiql are just some of the many open-source things available for use. Also check out IDE integrations like JS graphql.
- Deferred Resolvers give us a real way to attack the n+1 problem, which can be hard to solve in traditional REST.
- Deprecate API's on a field level. Client receives a deprecation whenever a field is marked to be deprecated. Once all the client dependencies are updated/removed the field can be safely deprecated.
- Great ergonomics. Write all your code in the same file; queries go right next to your markup. Need more data? Add it to your query and you're done.
- Very rapid prototyping and iteration when working with types (schema objects) that already exist in the GraphQL ecosystem.

Support for serverless applications. Running the GraphQL backend (except Subscriptions) on a serverless cloud function works really well. Since GraphQL exposes a single endpoint, you can run your entire GraphQL server off a single cloud function. Recently, the authors complemented and finished this formalization by proving that evaluating the complexity of GraphQL queries is a NL-problem [5].

Vogel et al. [6] present a case study on migrating to GraphQL part of the API provided by a smart home management system. They report the runtime performance of two endpoints after migration to GraphQL. For the first endpoint, the gain was not relevant; but for the second, GraphQL required 46% of the time of the original REST API.

Wittern et al. [7] also perform a study on GraphQL schemas. The authors study the design of GraphQL interfaces by analyzing schemas of 8,399 GitHub projects and 16 commercial projects. The authors report that a majority of GraphQL APIs have complex queries, posing real security risks

Vargas et al. [8] perform a study to investigate the feasibility of using a classic technique to test generation in GraphQL schemas (deviation testing). They use an implementation of GraphQL for Pharo and run the proposed technique in two popular GraphQL APIs.

4. GRAPHQL DRAWBACKS

There are many use cases where it's quite appropriate to GraphQL, but here are some instances where it's not -

- Authentication is usually handled by headers / hashing (stateless) or by specific service endpoints that then set up authenticated sessions or provide tokens for use with the API.
- File Uploads has been a major pain point in GraphQL. You could send a base64 string with a mutation, but large files result in large (and therefore unwieldy, slow to process) strings. A dedicated endpoint for uploads is more practical. Another option can be multipart request extension GraphQL mutations as explained here.
- Dynamic Connections. This can happen with generic key/value pairs where the value amounts to a foreign key. The nature of those connections will vary from query to query.
- RPC-Style Operations. Mutations are expected to execute a query and return those results. This can be unnecessary overhead for some asynchronous RPC-style operations.
- Caching can get difficult since the requested fields with each query can change; it uses a single endpoint which can return different kinds of payloads vs REST where there are multiple endpoints, and response payload remains constant. The problem is partially solved by using persisted GraphQL queries that assist in producing a JSON file for mapping queries and identifiers.
- Potential for a single point of failure. Since the entire resource (and its fields) are defined on a single endpoint, any breaking change on the type

/ endpoint will negatively affect multiple client apps consuming it.

- Since GraphQL is *not* a versioned API the process for designing additions to the graph is more rigorous. GraphQL server implementations do support @deprecated directive OOB which helps.
- Overkill for small apps. GraphQL is a good fit for complex systems with multiple microservices, but for simple apps, it might be better to go with the tried and tested REST architecture.

One important thing to keep in mind is that by opening up the ability for clients to query across the domain space and relationships, you add the possibility of very complex data loading. GraphQL gives a client complete freedom to request whatever it needs. This can get contentious since the client can potentially request many fields across multiple resources and thus causing sluggish network calls. This is similar to how queries in SQL can get very complex and expensive. Add in the recursive relationships that can exist in the graph and some queries can tax your system.

5. KEY TAKEAWAYS

While GraphQL is an extremely powerful and flexible API strategy, it is not a *silver-bullet* for all your data CRUD needs. You should evaluate your application needs and developer skills to make the right call. GraphQL adoption (switching from REST) usually requires a major rewrite of the API and Client layer for your application. While there are material benefits to switch, depending on the size and complexity of your app, this can be a massive undertaking in terms of time and resources. There is also a learning curve with GraphQL and its best practices which should be taken into account before taking the leap.

At the same time, GraphQL can remarkably simplify/optimize your data access and modification needs for both client and server-side engineers, regardless of languages or environment you're in. If you're writing an app from scratch and/or not afraid to try something new, GraphQL presents itself as a great option with many compelling reasons to use it. If you decide to adopt GraphQL, take enough time to design your graph schema. *Measure twice and cut once*. Mapping a good GraphQL schema is a non-trivial task. So, take your time and try to get it right the first time around; it will save you from a lot of inconvenience down the line.

6. CONCLUSIONS

GraphQL shifts your focus, from thinking of data in terms of resource URLs and traditional REST endpoints, into a graph of objects and the models used in apps. You can read more about this in the Facebook blog that was published back in 2015.

GraphQL provides some significant benefits over RESTful architecture. It can substantially simplify and optimize your data retrieval and modification requirements while allowing engineers to deliver faster. Having said that, it is not a panacea for your data access needs and the *devil is in the detail*. If chosen for the wrong reasons or not implemented correctly, it can negatively affect your application and developer experience.

REFERENCES

[1] Anmol Gaba, Dr. G N Srinivasan "Decomposition and Modularity in Software Systems" in International Research Journal of Engineering and Technology Volume: 07 Issue: 06 - June 2020.

[2] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture", *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115-150, 2002.

[3] G. Brito, T. Mombach, and M. T. Valente, "Migrating to GraphQL: A practical assessment," in 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 140–150.

[4] N. Schrock, *GraphQL introduction*, [online] Available: https://reactjs.org/blog/2015/05/01/graphql-introduction.html.

[5] O. Hartig and J. Pérez, "Semantics and complexity of GraphQL", *27th World Wide Web Conference on World Wide Web (WWW)*, pp. 1155-1164, 2018.

[6] M. Vogel, S. Weber, and C. Zirpins, "Experiences on migrating RESTful Web Services to GraphQL," in 15th International Conference on ServiceOriented Computing (ICSOC), 2017, pp. 283–295.

[7] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, "An empirical study of GraphQL schemas," arXiv preprint arXiv:1907.13012, 2019.

[8] D. M. Vargas, A. F. Blanco, A. C. Vidaurre, J. P. S. Alcocer, M. M. Torres, A. Bergel, and S. Ducasse, "Deviation testing: A test case generation technique for GraphQL APIs," in 11th International Workshop on Smalltalk Technologies (IWST), 2018, pp. 1–9.

BIOGRAPHIES

Amit Jambusaria - MS Computer Engineering from Syracuse University, NY, USA. Mr. Jambusaria is a seasoned technical leader and independent researcher with 11+ years of experience in the field. SME in areas such as GraphQL, Front-end web platforms and frameworks, Container technology, Cloud computing, Kubernetes (k8s) Microservices, APIs, UX/UI and Web performance.