Decision Guide to GraphQL Implementation

March 2020

Notices

Notices Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents AWS's current product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS's products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied. AWS's responsibilities and liabilities to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

© 2020 Amazon Web Services, Inc. or its affiliates. All rights reserved.

Decision Guide to GraphQL Implementation

Development teams are increasingly choosing to build their APIs with GraphQL. In this guide we first provide an overview of the benefits of GraphQL to help you determine if GraphQL is right for your organization. We then walk through what to consider when implementing a GraphQL API and whether you should host your own GraphQL server or choose a fully managed solution. Let's get started.

What is GraphQL?

GraphQL is a query language for APIs that was originally developed by Facebook and open-sourced in 2015. Quoting from the GraphQL Foundation:

> "GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools."

How does GraphQL work?

GraphQL provides users the flexibility to define which fields, from which data sources, they would like to request in a query. It does so by defining the shape of the output within the query itself. This feature results in the characteristic look of GraphQL queries, which mimic the responses to the queries.

A typical GraphQL query can look like this:

```
query {
 post(title: "How does GraphQL work?") {
  id
  author {
   name
   profile_url
  }
  content
 }
}
```

The result of such a query could be:

```
{
 "post": {
  "id": 123,
  "author": {
   "name": "Jeff Barr",
   "profile_url": "https://aws.amazon.com/blogs/aws/author/jbarr/"
  },
  "content": "GraphQL gives the API users the flexibility..."
 }
}
```

The query is sent as the payload to the API endpoint, and the result returns only the fields specified in the request.

A GraphQL API normally has a single endpoint to give you access to all the data sources you would need for a particular application or page. In the example above, a traditional API might have required that you request the details of the post from one endpoint, and the information about the author from another endpoint. But in a well-architected GraphQL API, these two entities are available through a single endpoint.

A GraphQL API has both a client and a server component. On the client side, it is common to use a library that implements the GraphQL standard in your programming language of choice. On the server side, a GraphQL API server listens to GraphQL API requests, parses them, and fetches the right data for each request from back-end services and data stores.

Why is GraphQL useful?

At the macro level, GraphQL is useful because it increases development speed and simplifies API management. GraphQL gives developers the ability to fetch just the data that's needed on each API call, and to fetch the data from multiple data sources. From the perspective of an API operator, GraphQL makes it easier to document and evolve an API.

GraphQL benefits for development teams

For development teams, using GraphQL provides faster time to market, improves developer productivity, and enables teams to scale APIs effectively – all of which increase development cycles and improve end-user experience. Specific reasons why development teams choose GraphQL are detailed in this section.

Frictionless development

When you work with a single API endpoint it means you can develop faster. Fetching an additional API field is as easy as updating your GraphQL query. Developers don't need to add another API client, understand the logistics of connecting to a new API server, or worry about choosing the right API version. Eliminating the fetching of unnecessary data is also straightforward—all it takes is a GraphQL query update.

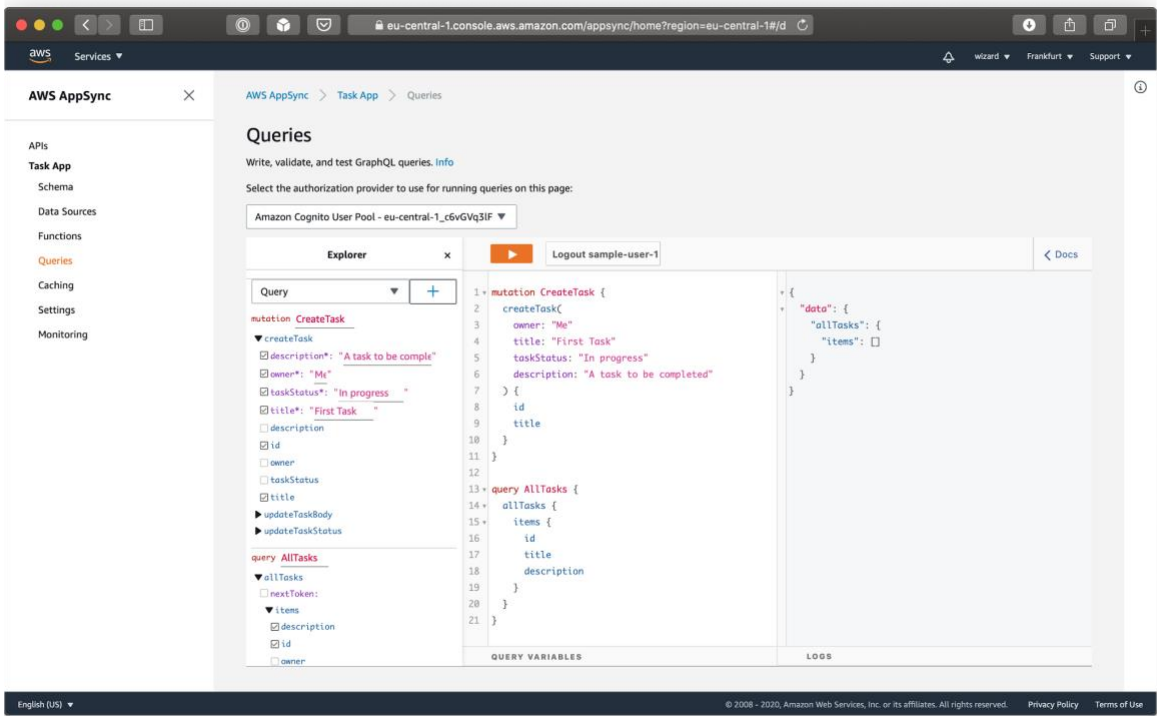
Data efficiency

When you get back only the data you need, it speeds up your development process. Developers don't need to spend extra time filtering out unwanted information or getting to objects contained deep in the structure of the JSON response. In addition, applications can vary the queries they use depending on screen size. On a larger screen, the user is likely to have a more stable internet connection, so the web app can request more data fields in the API response. The same app running on a mobile device can request less data and thus be more data-efficient.

Self-documenting

When you describe a GraphQL schema (the functionality available to the client applications that connect to it), it provides attribute names, data types, and description. This makes GraphQL inherently self-documenting. Every possible query, object, and field comes with a name, description, and types of information that can be queried from the server in a standard way. This means developers can onboard faster and easily explore the data available using manual queries or a tool like GraphiQL. GraphiQL is an interactive in-browser IDE that allows you to explore a GraphQL API endpoint, understand which entities are available, and try out the API in your browser.

In the image below you see the GraphiQL interface in the AWS AppSync console. AWS AppSync is a fully managed GraphQL API Service with a GraphiQL Integration in the AWS AppSync console. This lets developers easily see the details of the GraphQL API they're creating in AWS AppSync.



Improved page speed

With all API entities available through the same API endpoint you can fetch the entire API payload required for a particular page with a single API request. Fewer API requests mean quicker application load time for your end users. By defining the fields an API response should contain, developers can reduce the bandwidth required to download each API response. When you fetch just the data you need in an API request, you considerably speed up a web or mobile application.

GraphQL benefits for back-end development and operations teams

For operations teams, the advantage of GraphQL translates into a lower total cost of ownership for your API. Specific reasons why back-end development and operations teams choose GraphQL are detailed in this section.

Simplified API evolution

A key benefit of GraphQL for operations is the ability to expose the information from multiple back-end datastores in a single API. From the developer standpoint, it's much faster to use a single GraphQL API than to access every datastore directly. On the back-end, it is also easier to upgrade and change data sources without requiring developers to change the way they access the data.

Efficient monitoring and security

Monitoring and API analytics code can be embedded at the GraphQL API level thus ensuring that all metrics and monitors are consistent across GraphQL entities. It's also easier to maintain the security measures at the API layer—all underlying services get the security improvements with little to no work required by the service teams.

Effective caching and simplified real-time subscriptions

Another advantage of GraphQL for operations is the low number of endpoints—it's a GraphQL best practice to offer just one endpoint. Directing all API traffic to flow through one location facilitates caching and makes it easier to manage high-traffic APIs as well as real-time use cases like subscriptions. Modern GraphQL servers allow developers to cache complex responses as well as parts of larger responses. Effective caching is the key to keeping the load on the back-end system manageable and improving the performance of the API at the same time.

Streamlined maintenance

From the organizational perspective, each backend data source can be maintained by a different engineering team. A common operational need is to have clear ownership of the different parts of the business logic, and this logic can span across multiple codebases. Lack of clear ownership can sometimes mean delays in fixing critical issues and more operational overhead. With a GraphQL API, teams are able to implement fixes and features in the relevant parts of the API without slowing down other development teams.

Considerations for GraphQL implementation

It makes sense to approach a GraphQL API implementation for your web or mobile application differently based on whether you're building a greenfield project or adapting a legacy codebase. Here are two implementation approaches you can take.

Create new APIs

Often organizations choose to start their GraphQL journey with a new, "greenfield" API. In many cases they begin with an internal, non-mission critical API and then expand to mission-critical use case as they grow their competence with GraphQL.

When starting with a new API, focus on (internal or external) customer journeys and the profiles of your future API users. If your main customers are internal front-end and mobile developers who will build a user interface, think about the pages they will build and in what sequence. Also consider the amount of work your end user would require to start using your API. If they're already using a traditional API and new functionality is only available through GraphQL, your users might be frustrated to have to use two APIs in parallel. However, a net new user who is not using your legacy endpoints would require the same amount of work to start using a legacy API versus a GraphQL API, so they would be more likely to adopt your GraphQL endpoints.

Abstract existing APIs

Another way organizations start with GraphQL is by abstracting existing APIs. For a set of existing APIs, abstracting a number of endpoints into a single GraphQL API delivers GraphQL advantages without the work of migrating APIs or rewriting the application from scratch.

To identify the endpoints that could benefit from a GraphQL abstraction, consider the usage patterns for the existing endpoints. Having usage stats for the existing API will help with this task. Which of the endpoints are frequently used together? For example, is an application always loading a set of API resources for a single page? If so, creating a single GraphQL endpoint can help you merge those multiple requests into one request. Once your application correctly consumes that GraphQL endpoint, you can unlock faster page load times.

If you don't have access to such data, consider investing the extra time in implementing API usage tracking before diving into the abstraction work. Evaluating reliable statistics will help you identify the most promising areas in which to implement GraphQL and help you measure progress.

Choosing a GraphQL server

To allow your client application to query your GraphQL API you will need to have a GraphQL server. A GraphQL server is the interface between your backend and your application front-end. It parses queries written in GraphQL and fetches the data from connected databases or microservices. Here, you connect the API entities with their respective backend services. This logic has two components: schemas and resolvers.

The schema defines the shape of your data: its types and access permissions, and the relationships between different data entities. Resolver functions specify how to retrieve data from these sources; all of your GraphQL API's complexity resides in these functions. That means they're hidden from your client, who can only access your API via a single endpoint. This makes for a clean, easily manageable interface.

When choosing a GraphQL server, you have to decide between hosting it in your own environment with an open source GraphQL server or using a fully managed service from a SaaS or cloud provider. The following section walks through the pros and cons of each option.

GraphQL self-hosted open source implementation considerations

The main advantage of self-hosting an open source GraphQL server is flexibility. You can select the open source project that includes features that most closely align with your needs, and have the option to contribute code to the project to add or extend features. You also can implement and operate your GraphQL server on the infrastructure of your choosing. This gives you the ability to finely tune the operational characteristics of your API.

This level of flexibility does, however, have its downsides. With open source solutions API developers have to spend time writing non-business logic code to connect data sources, implement authorization and authentication, and integrate other common functionality such as caches to improve performance, subscriptions to support real time updates, and client-side data stores to keep off-line devices in sync. This means they have less time to focus on the business logic of their application. Similarly, back-end development teams and API operators of open source solutions need to provision and maintain their own GraphQL servers. As a result, instead of enjoying the simplicity and efficiency of a purely serverless model, operators are responsible for monitoring, scaling, and troubleshooting their API infrastructure.

If you decide self-hosting is the best avenue for your organization here are a few options; all of these open-source options work great on AWS.

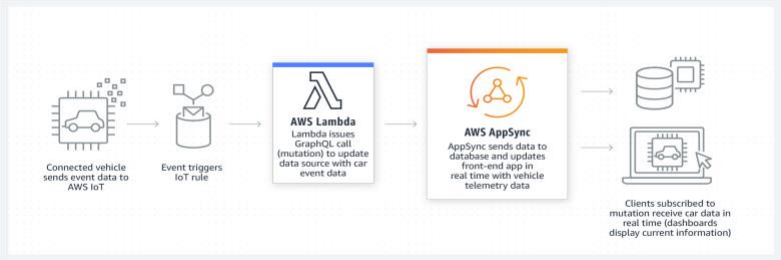
- Apollo provides convenient tools for creating a project with GraphQL. Among them are Apollo Clients for multiple platforms and Apollo Server for serving data using GraphQL. They are among the most popular tools for building JavaScript applications with GraphQL.
- graphql-ruby is a Ruby library for implementing a GraphQL server.
- Juniper is a GraphQL library for Rust.
- gqlgen is a Go library for implementing a GraphQL server, developed by 99designs.
- Lacinia is a GraphQL implementation in Clojure, developed by Walmart.

GraphQL managed service implementation considerations

The primary advantage of choosing a managed service for your GraphQL server is that it can be set up quickly and, because managed services are most often "opinionated" and optimized for a particular set of use cases (e.g. connecting with multiple AWS data sources, real-time subscriptions, off-line data synching) increase your development velocity by reducing the amount of non-business logic code you need to write for your API. While you will lose some of the flexibility available from a self-hosted, open source solution, you can gain simplicity and speed.

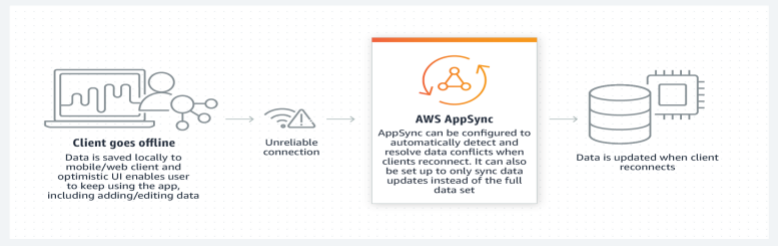
One such fully managed service is AWS AppSync. With AppSync, developers can easily use data sources like AWS DynamoDB, Lambda, Aurora, Elasticsearch, or any other data source that can connect via HTTP without having to separately configure or manage the connections. Developers can easily write business logic against these data sources by choosing from code templates that implement common GraphQL API patterns. Developers can easily interact with other AppSync functionality such as caching to improve performance, subscriptions to support real-time updates, and client-side data stores that keep off-line devices in sync. Below are examples of use case types and how they work when using GraphQL with AppSync.

Real-time Applications: (IOT as an example)



Unified Data Access

Offline Application Sync



From an operator's perspective, AppSync provides a fully managed GraphQL server, which means operators can enjoy the simplicity and efficiency of a purely serverless operating model. Similarly, integrations with AWS

CloudWatch for metrics and logs, AWS X-Ray for tracing, and AWS CloudTrail for audit logs make it easy to troubleshoot an API.

Tips for a successful GraphQL implementation

Successful GraphQL API implementation consists of more than just choosing the right server implementation. This section provides some tips to help you navigate your implementation of a GraphQL API.

Understand the problem you're solving

Before your team invests time into choosing the right GraphQL server, consider your particular use case. Here are some questions that can get you thinking:

- Who is going to use your GraphQL API? What are the needs of the internal and external API users?
- What is the query structure you want to offer? Which entities will be available through the GraphQL API?
- Which data sources are going to be required for the API to work? How can these data sources be accessed?

Make sure that your engineering and product teams agree on the approach you'll take in these areas. If your teams understand the approach from the outset it will lead to a smoother implementation.

Plan out the areas of ownership

A GraphQL API can unlock value across application domains, and this means that some parts of the value stream might not have a clear owner. A GraphQL API with a large number of entities might require greater coordination between teams. The different teams will also need to agree on how to resolve API enablement concerns from caching to endpoint discovery.

The following questions might be helpful in clarifying the areas of ownership in your future GraphQL API:

- Which teams are going to be responsible for the different sections of the GraphQL API? How will ownership evolve over time?
- Who is responsible for maintaining the API layer itself? Is the responsibility shared between the teams owning the individual API endpoints, or is there an enablement team?
- How can teams find out about new GraphQL API entities and endpoints? Who is responsible for maintaining API documentation?

Use GraphQL where it can add value

GraphQL isn't a silver bullet. If your REST or SOAP API is stable and does its job well enough, think about what you're trying to achieve by replacing it with a GraphQL API. Is the value added by GraphQL (e.g., the ability to query multiple API entities in a single endpoint) enough to justify the implementation work and user migration? Keeping the legacy endpoints in place can be the simpler and more economical solution in some cases.

GraphQL customer implementation case studies

ALDO Group

ALDO Group is a Montreal-based footwear and accessory retailer with a global presence. The company currently has 3,000 stores in more than 100 countries.

The ALDO in-store mobile app was very successful and helped store employees find and deliver shoes to the front of the store. Most retrievals are performed in under a minute, and given that 50,000 people walk into ALDO stores each day, the in-store application has high requirements for stability and speed. However, the mobile app's API and database layers used different technologies, and system outages would sometimes result in application downtime and frustrated customers.

ALDO Group chose to migrate their APIs to GraphQL to improve the stability of their services and allow their mobile developers to create customer value faster. The ALDO Group's team chose a managed GraphQL solution, AWS AppSync, to further improve the stability of the API and reduce the maintenance and scaling load on their development teams. Learn more about the ALDO Group's experience with GraphQL in the ALDO Group case study.

HyperTrack

HyperTrack is a cloud platform that offers location tracking as a service for mobile apps. Developers add HyperTrack's SDK to their mobile applications to have functionality like recording of live location data to the cloud, route tracking, location-based notifications, and more.

The team at HyperTrack chose GraphQL as an API for the data needed in their front-end applications. "We needed the ability to support GraphQL in our pipeline for both front-end and mobile teams to consume," says Thomas Raffetseder, Software Architect at HyperTrack. "Instead of having to use our own proprietary format, we like what GraphQL provided as a standard." HyperTrack opted to use a GraphQL managed service to minimize operational overhead. Learn more about HyperTrack's choice to use AWS AppSync for their GraphQL implementation in the HyperTrack case study.

Summary

In this article, we covered GraphQL basics, described how it can benefit your business, and covered considerations on how to implement GraphQL. We hope this resource helps you on your GraphQL journey.

Learn more about GraphQL and AWS AppSync on the AWS AppSync resources page.