GraphQL at Enterprise Scale

A Principled Approach to Consolidating a Data Graph

APOLLO

GraphQL at Enterprise Scale

A Principled Approach to Consolidating a Data Graph

Jeff Hampton

Michael Watson

Mandi Wise

APOLLO

GraphQL at Enterprise Scale

Copyright © 2020 Apollo Graph, Inc.

Published by Apollo Graph, Inc.
https://www.apollographql.com/

Revision History for the First Edition

2020-09-11: First Release
2020-10-27: Second Release
2020-12-10: Third Release

Contents

The Team

This guide is the culmination of thousands of hours Apollo's employees have spent working with and learning from our customers over the years.

These are the members of the Apollo team who have made contributions to the content in this guide:

- Jeff Hampton – Writing
- Michael Watson – Writing
- Mandi Wise – Writing/Editing

Preface

The **data graph** has quickly established itself as an essential layer of the modern application development stack. In tandem, GraphQL has become the de facto technology for managing this new layer with its enticing promise to bring together all of an organization's app data and services coherently in one place. And thanks to the wellspring of experimentation and innovation with GraphQL over the years, it has proven itself a mature and capable technology that's ready for scalability.

GraphQL makes its way into an enterprise's tech stack through a variety of avenues, for instance, a single team eager to leverage its client-driven approach to data fetching. However, as its adoption spreads realizing GraphQL's promise at scale requires coordination and **consolidation** of these efforts across teams.

At Apollo, we've had the opportunity to work with countless developers in a wide range of enterprises over the years. Through that work, we've learned that a unified, federated data graph is at the heart of any successful GraphQL consolidation project. We first shared some of these insights in Principled GraphQL where we outlined best practices that organizations can follow to create, maintain, and operate a data graph as effectively as possible. In this guide, we'll provide a detailed road map for putting these principles into action at the enterprise level.

Who Should Read this Guide

This guide is for engineering leaders. If your enterprise is currently using GraphQL, then you have undoubtedly experienced challenges related to maintaining a monolithic data graph or wrangling multiple smaller graphs. Consolidating GraphQL in your organization can help reduce friction points between teams, enhance developer experience, improve governance of your graph, and even provide better observability of how your data is consumed.

This guide is for business leaders. Consolidating your data graph isn't just about the architecture of your tech stack. It's about an organizational transformation that will harness the power of graphs to unlock platform value. A unified data graph increasingly lives at the center of value delivery in an enterprise and the strategies and tactics presented in this guide provide a pathway to realizing the potential of your data-graph-as-a-product.

This guide is for developers and architects. Whether you're a developer on a client team or actively maintaining a GraphQL server in production now, the concepts outlined in this guide will give you a clearer understanding of how your work can align to your organization's broader GraphQL strategy and even become a "Graph Champion" on your team.

What You'll Learn from this Guide

This guide is the culmination of what we've learned after spending thousands of hours working with enterprises at Apollo. Based on those experiences, we'll cover both the high-level considerations and the practical skills required to successfully consolidate a data graph across an enterprise.

We'll first present a case for why GraphQL consolidation is important in an enterprise and provide a framework for assessing when an enterprise should undertake a consolidation project. Subsequently, we'll move into the specifics of successful enterprise-level graph management and discuss the essential role of the graph champion in a consolidated GraphQL architecture.

Ultimately, this guide has been written for you by Apollo to help guide you on your journey toward effectively scaling your data graph across your enterprise. It's intended to be a living document and the solutions team will add additional content to it on an ongoing basis in future releases.

You can check for update and download the latest version of the guide here:

http://apollographql.com/guide

How to Contact Us

We'd like to hear from you if you have questions about this guide or have a unique perspective you'd like share about using GraphQL your organization. Email us at solutions@apollographql.com to reach out at any time with your comments or if you require any assistance implementing GraphQL in an enterprise environment.

Moving Toward GraphQL Consolidation

By Jeff Hampton and Michael Watson

This chapter will introduce you to the notion of creating a unified, federated data graph in an effort to leverage the benefits of a consolidated GraphQL architecture. Based on our experience working with a variety of enterprises at Apollo, we'll provide a rationale for consolidation as well as a framework for determining whether your organization is ready to consolidate its data graph.

Why Consolidate Your Data Graph?

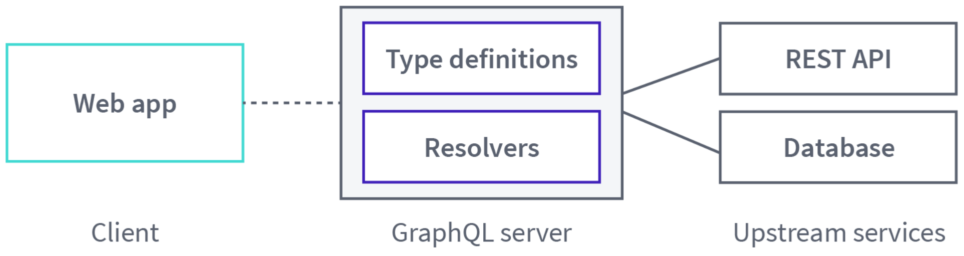
The GraphQL community and ecosystem of related software have grown at breathtaking speed. During the intervening years since its public release in 2015, this technology has quickly matured to a point where it can be used in nearly any infrastructure.

Companies such as Airbnb, GitHub, and the New York Times have famously already adopted GraphQL in their tech stacks. With its strong type system and declarative approach to data-fetching, it's easy to see why teams across enterprises have been eager to embrace the many benefits of GraphQL. At Apollo, we see firsthand the level of enthusiasm organizations have for GraphQL with over 1.5 million downloads of the Apollo Client packages every week, along with hundreds of thousands more weekly downloads of the Apollo Server and Apollo Federation packages.

Scanning your organization you may quickly realize that multiple teams are already using GraphQL in production today. Having some top-level insight into how GraphQL is used across your enterprise is the first step toward understanding whether those efforts can and should be consolidated.

How GraphQL Gains Traction in an Enterprise

When developers begin to experiment with GraphQL, they almost invariably first encounter a foundational architecture where a client application queries a single GraphQL server. In turn, the server distributes those requests to backing data sources and returns the data in the client's desired shape:

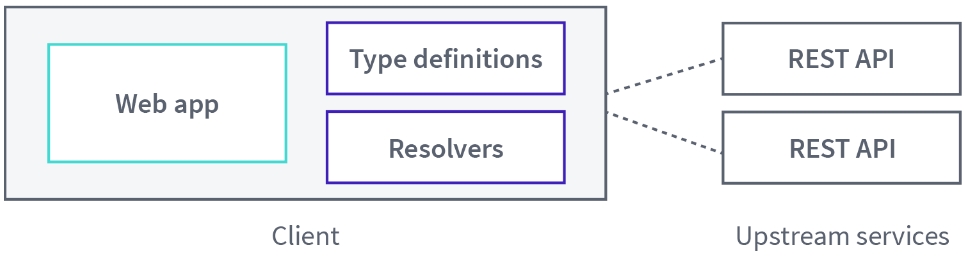


As different teams within an enterprise move toward officially adopting GraphQL, the complexion of their isolated implementations will usually be adapted from this basic architecture, but may vary considerably from from team to team. At Apollo, we've typically seen that those initial, unconsolidated efforts resemble one of the following four patterns.

Pattern 1: Client-Only GraphQL

Client teams that are enthusiastic to reap the benefits of GraphQL's client-centric data-fetching capabilities may charge ahead and implement a GraphQL API within the context of their application. With such implementations, these teams are often motivated to adopt GraphQL for the convenience of wrapping existing APIs with a single GraphQL API endpoint.

To illustrate this approach, a client-only GraphQL architecture may look like this:

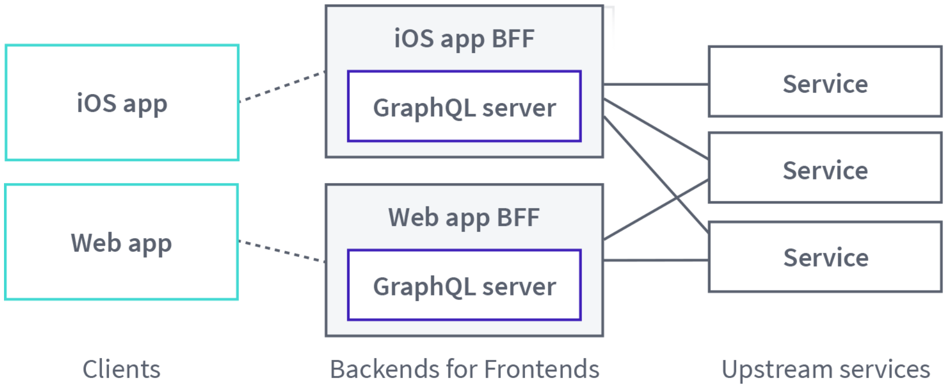


Pattern 2: Backend for Frontend (BFF)

GraphQL may also be used as a solution for teams implementing the Backend for Frontend (BFF) pattern. BFF seeks to solve the problem of requiring different clients (for example, web and iOS) to interact with a monolithic, general-purpose API. Alternatively, BFFs can save client applications from making requests to multiple backend services to obtain all of the data required to render a particular user interface view.

As a solution, BFFs add a new layer where each client has a dedicated BFF service that directly receives the client's requests and is tightly coupled to that user experience. For teams creating BFF services, GraphQL can be a natural fit for building out this intermediary, client-focused layer and adopting this pattern can be an important first step toward consolidating a data graph.

In practice, the BFF pattern with GraphQL may look like this:



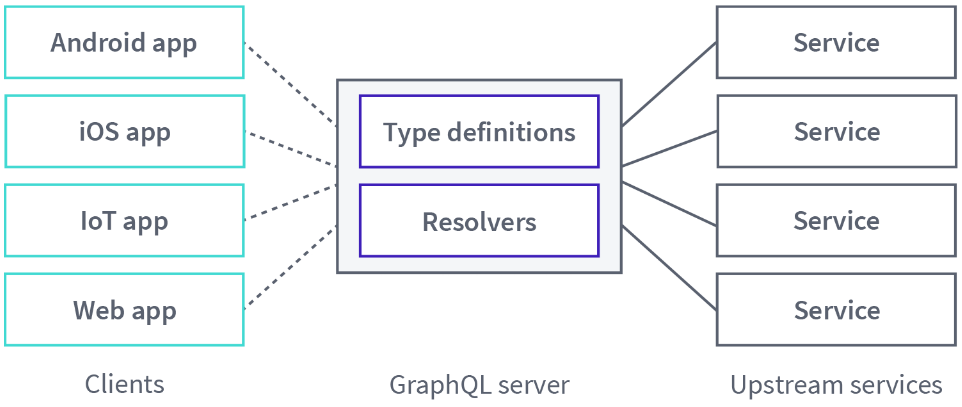
Pattern 3: The Monolith

The monolith pattern can take on two forms in an enterprise. In its first form, teams may share one codebase for a GraphQL server that is used by one or more clients. In some cases, client code may even live in the same repository as the GraphQL server. However the code is organized, the ownership of this graph is shared by the various developers who ultimately consume the graph's data.

In its alternative form, a single team may be designated to own a graph that is accessed by multiple client teams. This team would typically define a set of standards for the graph and champion its adoption throughout the organization.

As with GraphQL-based BFFs, maintaining a single, monolithic GraphQL API can help set the stage for effective consolidation of an organization's GraphQL-focused efforts.

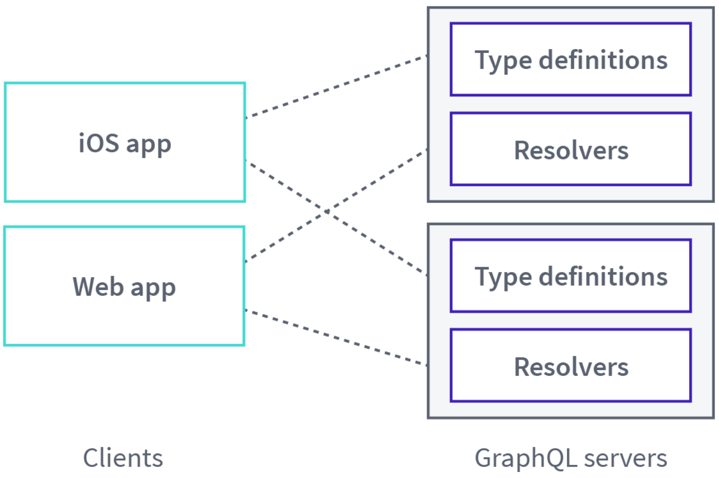
For either monolithic scenario, its high-level architecture looks like this:



Pattens 4: Multiple Overlapping Graphs

Enterprise teams may also independently develop their own service-specific GraphQL APIs in tandem. With this approach, teams may delineate each service API based on types or use cases, but there will often be overlap between the graphs due to the interconnected nature of data.

Such an architecture may look like this:



Where Do These Patterns Break Down?

After taking stock of who uses GraphQL and how in your enterprise, the patterns the various teams have implemented can provide insight into what kinds of problems they initially endeavored to solve. Similarly, these choices can help illuminate what pain points the teams currently face with respect to how GraphQL is used in their tech stacks.

Client-Only GraphQL

Teams that opt for client-only GraphQL approaches are motivated to improve their client development experience by layering GraphQL on top of the REST endpoints or other legacy APIs they have to work with. And while improved developer experience is a win, beneath this abstraction the client application will still incur performance costs as it maintains responsibility for making multiple requests to various services to gather all of the data required to render a view.

BFFs

Like client-only approaches, teams that use GraphQL with BFFs enjoy the advantage of improved developer experience by way of a consumer-friendly GraphQL API, but they also manage to overcome the performance issues incurred by client-only approaches. BFFs accomplish this by providing a unified interface for a client application to send its requests while also handling the heavy lifting of querying multiple backend services on behalf of the client.

However, there is an inherent tradeoff in building and maintaining BFFs. When every client team is empowered to create a BFF to suit their needs, there will be inevitable duplication of effort across those teams. However, where BFFs are shared between seemingly similar clients in an effort to reduce duplication, then the GraphQL schema contained within can balloon in size and become confusing due to the lack of clear ownership.

Monoliths

The pains that emerge from shared BFFs are only sharpened with monolithic GraphQL server implementations that have shared ownership. Portions of a graph may be well-designed to suit the needs of certain client teams only, while other clients must find workarounds or create overlapping types for their own use. Correspondingly, standardization becomes an issue because the shape of the graph evolves myopically on a client-by-client or a feature-by-feature basis.

Even in scenarios where a dedicated server team maintains ownership of the graph challenges quickly arise when more than one graph definition is required for a single product in order to support the needs of multiple clients. A server team may also find itself burdened with the task of building and maintaining the necessary tooling to evolve the schema over time to meet new product needs without breaking compatibility for any clients that are actively consuming data from the graph.

Multiple Overlapping Graphs

Finally, when multiple graphs exist within an enterprise it often indicates that the organization was an early adopter of GraphQL, moved to production quickly, and invested more in GraphQL as time went on. As one potential outcome of this investment, an attempt to expand a monolithic GraphQL API across teams may have ultimately resulted in the graph being split into multiple pieces to accommodate the conflicting needs of each team. The inevitable result of this approach is a duplication of effort to manage these two overlapping graphs and a subpar experience for client applications that no longer have a unified interface from which to request data.

Another possible reason an enterprise may have multiple overlapping graphs stems from a deliberate choice for teams to manage their GraphQL APIs independently but assemble them into a single gateway API using schema stitching. While schema stitching can simplify API usage from a client's perspective, the gateway API requires a considerable amount of imperative code to implement. What's more, it may not always be clear-cut where to split types across services and it also necessitates the designation of an API gatekeeper who will manage the gateway and how the underlying schemas are composed into it.

Inconsistency: The Common Shortcoming

All of the previous patterns—whether client-only GraphQL, BFFs, monoliths, or multiple overlapping graphs—also have a shared shortcoming in that their implementations result in a **lack of consistency.** A more productive way forward for teams searching for better efficiency and understandability from their GraphQL-based architectures will have two requirements:

1. **Consumers should be able to expect consistency in how they fetch data.** A single endpoint should be exposed to client applications and, regardless of what underlying services supply the data, clients should be able to use consistent workflows to consume the data.
2. **Providers should consistently represent common entities in a consumption-friendly way.** Teams may be empowered to use any underlying technology at the data layer, but access to this data should be consolidated through the GraphQL API and exposed in a way that compliments client use cases. Additionally, teams should be able to delineate service boundaries based on separation of concerns (as opposed to separation by types) without interfering with each other.

How Consolidation Addresses These Challenges

Consolidating your data graph is the key to moving beyond these architectural pitfalls, achieving consistency, and realizing the full potential of GraphQL in an enterprise.

At a fundamental level, moving toward graph consolidation requires that your organization has **one unified graph** instead of multiple graphs created and managed by each team. However, the implementation of that unified graph should be **federated across multiple teams**. These are the first two "integrity principles" outlined in Principled GraphQL.

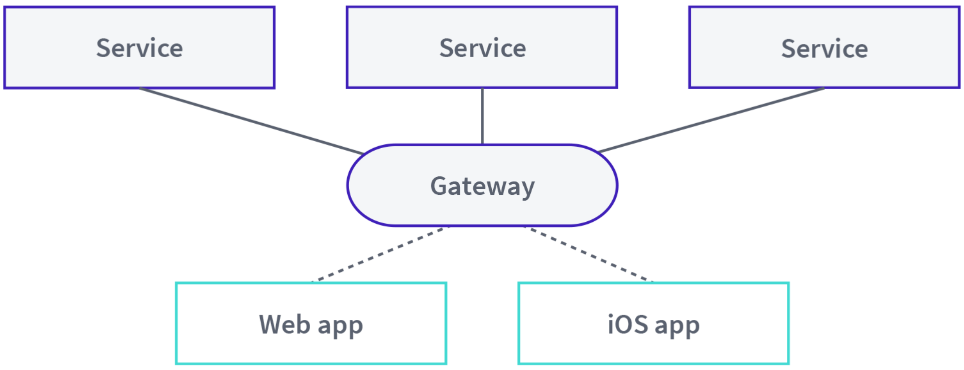
Specifically, moving toward this kind of consolidated data graph allows teams across the enterprise to:

- **Scale GraphQL APIs effectively.** Implementing uniform practices allow the benefits of GraphQL to be realized at scale in an organization. For example, teams will have a better understanding of the workflows and policies that they must follow to make contributions to the graph. Similarly, they will also benefit from improved standardization when consuming data from the organization's graph.
- **Obtain a unified view of your data.** Your graph is a representation of the data of your product. Having a consolidated view of this data will provide you with fresh perspective into how that data is currently used, while also inspiring new creative uses for it in the future. Additionally, it will help you to enforce a measure of consistency on how client applications consume that data.
- **Leverage existing infrastructure.** GraphQL consolidation allows teams to reuse existing infrastructure in an organization and help eliminate duplicated efforts where teams interact with data. Consolidation also allows you to take a holistic view of the practices and tooling developed by each team that touches your data graph and leverages the best of those individual efforts across the enterprise as a whole.
- **Ship code faster.** Organizations adopt GraphQL to build and iterate on their products faster. As GraphQL gains traction throughout an enterprise, these benefits may be partially offset by time spent developing tooling to help support that growth. Consolidation helps reclaim that lost momentum by providing a clearly defined set of practices for teams follow when contributing to or consuming data from the graph.

What Does a Consolidated Data Graph Look Like?

In practice, a consolidated, federation-driven GraphQL architecture consists of:

- A collection of **implementing services** that each define a distinct GraphQL schema
- A **gateway** that composes the distinct schemas into a **federated data graph** and executes queries across the services in the graph



Apollo Server provides open source libraries that allow it to act both as an implementing service and as a gateway, but these components can be implemented in any language and framework. Specifically, Apollo Server supports federation via two open-source extension libraries:

- `@apollo/federation` provides primitives that your implementing services use to make their individual GraphQL schemas composable
- `@apollo/gateway` enables you to set up an instance of Apollo Server as a gateway that distributes incoming GraphQL operations across one or more implementing services

We will cover consolidated GraphQL architectures using Apollo Federation and Apollo Gateway in-depth in Chapter 3.

Unlike other distributed GraphQL architectures such as schema stitching, federation uses a declarative programming model that enables each implementing service to implement *only* the part of your data graph that for which it's responsible. With this approach, your organization can represent an enterprise-scale data graph as a collection of separately maintained GraphQL services. What's more, schema composition in federation is based on GraphQL primitives, unlike the imperative, implementation-specific approach required by schema-stitching.

Core Principles of Federation

A GraphQL architecture that has been consolidated with federation will adhere to these two core principles:

Incremental Adoption

If you currently use a monolithic GraphQL server, then you can break its functionality out one service at a time. If you currently use a different architecture like schema stitching, then you can add federation support to your existing implementing services one at a time. In both of these cases, all of your clients will continue to work throughout your incremental migration. In fact, clients have no way to distinguish between these different data graph implementations.

Separation of Concerns

Federation encourages a design principle called *separation of concerns*. This enables different teams to work on different products and features within a single data graph, without interfering with each other.

By contrast, traditional approaches to developing distributed GraphQL architectures often lead to **type-based separation** when splitting that schema across multiple services. While it may initially seem straightforward to divide a schema by type, issues quickly arise because features (or concerns) managed by one service often span across multiple types that are located in other services.

By instead **referencing** and **extending** types across services, **concern-based separation** offers the best of both worlds: an implementation that keeps all the code for a given feature in a single service and separated from unrelated concerns, and a product-centric schema with rich types that reflects the natural way an application developer would want to consume the graph.

When to Consolidate Your Data Graph

At this point, you may have a sense that your enterprise could benefit from consolidating its data graph, so the next important question to answer is *when* should it move toward consolidation?

GraphQL, from a pure engineering standpoint, is one means to achieve a common set of business goals: horizontal scalability, rapid product iteration, and increased service delivery capacity, and reduced time-to-market. When placed in the hands of architects and engineering leaders, common questions emerge about how GraphQL can and will change the organization.

At a fundamental level, a conversation about consolidation can begin **as soon as it seems logical for multiple teams to manage different parts of the data graph.** While each organization and line of business may have unique considerations in answering the question of when and how to consolidate, Apollo has recognized patterns of success and failure when making this organizational shift. Additionally, any good architect should spend sufficient time laying the groundwork for future change. While it might be tempting to federate "early and often," consolidating through federation requires meeting a threshold and burden of evidence that the enterprise will benefit from this approach.

In the spirit of Principled GraphQL, we present a framework for making this decision, illuminating the potential gaps in an organization's success plan, and ensuring constant success throughout the organization's GraphQL evolution to a federated implementation.

With a process in place to answer this question and evaluate the capability of success, we'll explore some common scenarios taken from real-world projects here at Apollo. The real value of GraphQL lies in the hands of those tasked with its implementation, and organizations of all sizes and shapes face the same human-centric challenges with more or less success, and with more or less friction during the process.

To frame this decision-making process, we should first examine the inherent strengths of implementing or extending a federated data graph.

The Strengths of a Federated Graph Implementation

Just as network performance tuning is bound by the speed-of-light, the organizational optimizations offered by a federated GraphQL implementation is bound by some real-world constraints:

- **Consensus:** A collective understanding of data graph entities, tools, and quality
- **Responsibility:** Clear delineation of data graph "ownership," education, and support available to teams
- **Delivery:** Speed of infrastructure change, velocity of product delivery
- **Performance:** Impact to consumer-facing operation resolution for distributed operations

At its heart, a federated GraphQL implementation is an *optimization toward separation of concerns* (be they performance, team structure, delivery cadence, line-of-business alignment, or some combination of these) in exchange for a distributed system. The shift toward microservices also involved this tradeoff, but without a demand-driven, product-delivery orientation.

When deciding to break a monolithic graph into a federated one or when expanding a federated graph by adding new services and teams, an architect should have a plan for addressing the above four areas of concern. The decision matrix below is annotated with each of these concerns and provides guidance in resolving any gaps in measuring, understanding, and addressing these concerns.

Decision Framework Matrix

Whether you're adding a new service, splitting an existing service, or choosing to implement a federated graph for the first time, **an architect's most important responsibility is understanding the motivation for the change**. In Apollo's experience, a lack of clear and reliable measurements makes it harder to decide where and when to separate the concerns among graph services.

At a strategic level, GraphQL adoption and evolution to a federated implementation can be measured reliably using a simple matrix. By answering these questions periodically, technology leaders will have a continuous evaluation of when, and how, their GraphQL implementation should proceed.

Our recommendation is to keep this exercise simple and stable. Practitioners should use the **Apollo Consolidation Decision Matrix** below as a regular artifact to aid in a formal decision-making process.

If the answers to **all** of these questions are "yes," then you should proceed to laying out a clear path to a successful implementation.

If the answers to **any** these questions are unclear or "no," then leaders should take caution in evolving their GraphQL implementation to federation:

- Use each "no" to identify and monitor metrics and indicators that change is necessary
- Approach each "no" with a relentless desire to connect with the team(s) doing the work and understand how this becomes a "yes"

Apollo Consolidation Decision Matrix

Concern	**Criterion**	**Yes**	**No**	**Remediation/Guidance**
Consensus	Are multiple teams contributing to your graph?			If this is an initial federated implementation, identify your "Graph Champions" (see the next chapter) and establish education, review, and governance processes.

Concern	Criterion	Yes	No	Remediation/Guidance
Responsibility	Are contributions to your graph by multiple teams regularly causing conflicts with one another?			If teams are collaborating well together, consider the potential switching cost of diving teams or adding new teams.
Delivery	Is there a measurable slowdown or downward trend in GraphQL service change delivery?			If there isn't a measurable, negative impact to product or service delivery, consider the additional complexity and support for this change.
Delivery	Is there a concrete security, performance, or product development need to deliver portions of your existing schema by different teams or different services?			If consumers or internal stakeholders are not currently affected, consider revisiting the driving factors for this change.
Consensus	Is there a single source of governance for your GraphQL schema within the organization?			An initial Federated implementation, or an early expansion of Federation, are good opportunities to create support systems for education, consensus-building, governance, and quality control.
Consensus	Does your GraphQL governance process have a reasonably robust education component to onboard new teams?			Apollo has found that a robust education plan is a leading indicator of constant improvement and success.
Delivery	Is your existing GraphQL schema demand-oriented and driven by concrete product needs?			Changes driven by data-modelling or internal architectural requirements may not have an ROI when weighed against the costs of infrastructure and organizational change.

Concern	Criterion	Yes	No	Remediation/Guidance
Responsibility	Do you have a strong GraphQL change management, observability, and discoverability story, and do providers and consumers know where to go for these tools?			Graph administration and tooling such as Apollo Studio are key elements in a successful, organization-wide GraphQL initiative.
Consensus	Is your existing GraphQL schema internally consistent, and are your GraphQL schema design patterns well-understood by providers and consumers?			Dividing responsibility or adding new schema to your Graph without strong governance may exacerbate existing friction or product/service delivery challenges.
Performance	Can you be reasonably sure that the cost of additional latency, complexity, and infrastructure management will have a positive ROI when bound by business timelines and objectives?			Ensure that the requirements for separating concerns have a performance and optimization budget.

Ensuring Constant Improvement and Success

The outcomes of a GraphQL consolidation project should be measured against the original, documented drivers for the transition to a federated data graph. Aside from these measurements, certain actions and approaches must be undertaken to ensure that ongoing changes to the consolidated GraphQL architecture will be a success from a human and technology perspective.

For instance, teams may need to adopt new processes and practices to evolve shared types collaboratively and in such a way that provides consistency for current consumers of the data graph. Additionally, while an incremental cost, the infrastructure impact should be explored and verified against reference architectures during the project.

Because GraphQL can be an organizationally transformative technology, care should be taken to involve all stakeholders during the planning and implementation process of a federated data graph. As a result, education plays a key role in the success of federated implementations, which we will begin to explore in the next chapter.

Summary

Consolidating GraphQL APIs across the enterprise can help bring a much-needed measure of consistency to how this technology is implemented for both data graph contributors and consumers alike. Moving toward a unified, federated approach allows an organization to scale its GraphQL APIs, obtain new perspectives on its data graph, reuse infrastructure, and enable teams to ship code faster. When the time is right to move toward a consolidated data graph, enforcing proper separation of concerns in the underlying services will allow teams to continue to rapidly iterate while adhering to the constraints imposed by the federated implementation.

In the next chapter, we'll explore the topic of graph ownership within an organization as well as how to plan for the successful roll-out of a consolidated graph architecture with federation.

Graph Champions in the Enterprise

By Jeff Hampton

As we explored in the previous chapter, GraphQL adoption patterns can vary considerably within large organizations. In some instances, GraphQL is identified by architects and applied as an incremental pattern of API consolidation or mediation. Alternatively, GraphQL spreads organically among product teams looking to accelerate their delivery with the safety and support afforded by the GraphQL specification and community. Regardless of its inception, GraphQL adoption naturally grows beyond a single team's ability to reason about what is being developed in an enterprise.

Apollo's experience has revealed a consistent need for a specific skill set around GraphQL in an enterprise. To put it plainly—regardless of the investment model—GraphQL adoption will eventually generate the need for consolidation once two or more teams invest in a data graph. The enterprise's Graph Champions will be instrumental to this consolidation effort.

In this chapter, we'll further explore the concept of **the data-graph-as-a-product**, identify its customers, and explore the skills and products necessary to consolidate GraphQL within an enterprise. We'll then scope the responsibilities of Graph Champions and their role in organizational excellence and we'll explore each component of graph championship and data graph administration with key deliverables and approaches to address consolidation challenges.

The Graph Champion and Graph Administration

The size and shape of the Graph Champion role may be embodied in a few teams members, an architectural review board, or simply a cross-functional guild. Regardless of its shape, the Graph Champion works to ensure that contributors and consumers of an organization's graph get what they need from it.

In short, the Graph Champion views an organization's **data graph as a product with multiple customers.** From that perspective, Graph Champions understand that:

- Time-to-market is crucial to customer success
- Product quality is necessary for customer trust
- Educating customers is a key factor in making a product useful
- The data graph must have an ecosystem of tooling that serves all customers well
- The ergonomics exposed to graph consumers and conntributors must be aligned with industry standards

Four Key Responsibilities of the Graph Champion

At Apollo, we have commonly seen that the core responsibilities of Graph Champions in an enterprise are divided into four overarching areas:

Governance

Broad initiatives are best served by a team whose focus and value is well-understood across business units and organizational boundaries:

- Graph Champions are recognized as a **source of truth** for GraphQL within the organization
- With an increased altitude, Graph Champions can be **entrusted with the security of the graph** and its access
- Teams can rely on Graph Champions to bring **clarity to cross-cutting concerns** (for example "how do I reference an end-user?" or "how do we handle media, currency, and internationalization consistently in our products?")
- Establish and maintain deprecation and long-term-support (LTS) schedules based on end user and consumer demand for graph features

Health

Graph Champions support healthy, consolidated, and federated data graphs that have these key characteristics:

- Healthy adoption of a single, federated graph requires **rigor** in maintaining a **cohesive, easy-to-consume** graph surface
- **Service discovery** and **product development** depend on consistent document **documentation**, **style**, and **availability**
- Consumers can serve end users quickly because the federated graph has **consistent naming** and **logical organization**

- **Do not contain** highly-duplicative or deceptively-similar portions of the graph
- **Avoid confusion and friction** for consumers

Advocate

Graph Champions serve the interests of multiple customers and stakeholders through support and service by:

- **Defending the role** of the data graph to **business leadership**
- Providing **education to new customers** in the languages and parlance of the teams to which they belong
- **Onboarding** and facilitating discussions, RFCs, and architectural reviews

Equip

Successful “digital transformation“ strategies often under-prioritize engineering ergonomics and tooling. A successful Graph Champion equips each customer of the data graph according to their needs by:

- Providing and manage tooling for other teams to use and evolve the graph
- Establish common, polyglot patterns and sound practices for effective GraphQL use
- Supporting delivery systems, including integration, testing, artifact registries, and IDE tooling

Supporting Customers of the Data Graph

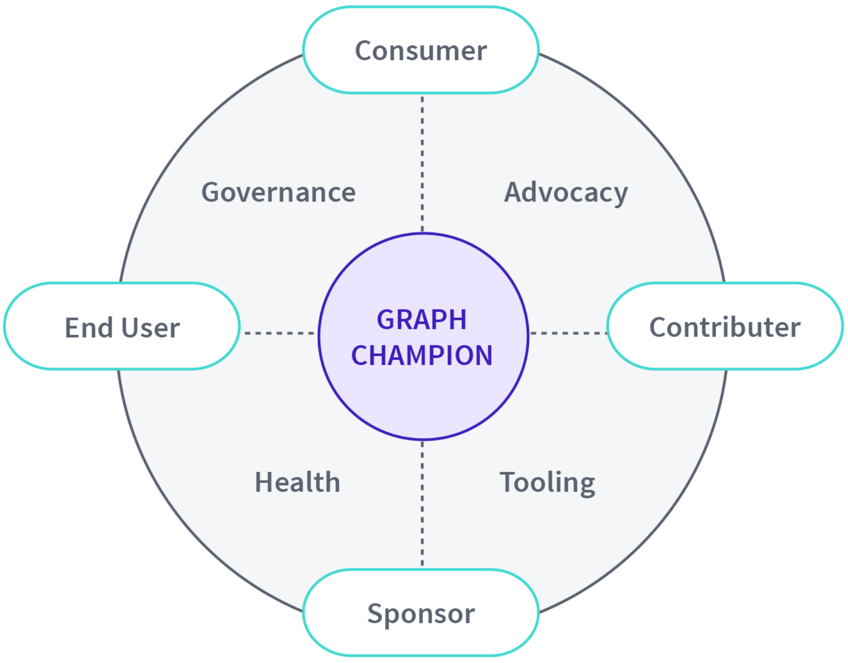
A new product-centric view of the data graph demands a clear understanding of the graph’s customers. Before moving forward, it’s important to recognize that and a customer-centric view of API service delivery is distinctly different from a stakeholder-centric view of an ongoing project. While stakeholders may bring concerns to a project’s lifecycle, customers bring feedback about how well the product supports them in achieving their goals.

To those ends, we have identified four unique customer personas that data graphs must support, each with different usage requirements and feedback perspectives to consider:

1. **End User**
    - Uses products built by the organization’s consumers
    - May use public APIs, cross-platform application experiences, or integration platforms
2. **Consumer**
    - Explores an organization’s graph

   - Builds products for End Users using existing and new graph features
   - Are concerned with performance, new product development
3. **Contributor**
   - Resolve graph data to underlying systems
   - Fulfill product-driven requests from Consumers
   - Collaborate with Consumers through tooling, education
4. **Sponsor**
   - Enable CI/CD and provide delivery platform
   - Maintain operational excellence
   - “Last Mile” to the End User

With these personas in mind, we can further contextualize the key responsibilities of Graph Champions from the previous section to gain a holistic view of their role in supporting a consolidated data graph in an organization:



Managing Consolidation Challenges

As organizations work toward consolidating their GraphQL service delivery through federation, a common set of challenges often arises as teams align to new practices for managing and contributing to a unified data graph. As an extension of supporting graph customers, Graph Champions can help an enterprise strategically address the following challenges:

Challenge #1: Schema Evolution

GraphQL increases developer velocity and product delivery. Graph Champions support this ongoing product evolution through education and governance so that the graph can continue to safely and effectively serve its customers.

Challenge #2: Composition

Different teams and business priorities frequently create blurred boundaries of domain, data, and service ownership. Graph Champions can facilitate domain-based conflict resolution of overlapping types, fields, and cross-cutting concerns in support of the broader health of the composed data graph.

Challenge #3: Service Delivery

Organizations delivering a data graph as a product must reason about services and schemas with different rates of change and different delivery timelines for end-user products. Graph Champions can help provide the necessary insight to configure service boundaries that allow one team to maintain and evolve its portion of the graph without compromising or otherwise conflicting with the work of other teams.

Challenge #4: Tooling

GraphQL devops has matured. Service delivery demands observability, performance tuning, and client/operation identification. Graph Champions act as advocates for proper developer and operational ergonomics to support teams in effective service delivery.

Delivering Organizational Excellence as a Graph Champion

There are some higher-level questions that can guide the mission and day-to-day and week-to-week work of the GraphQL Champions in an enterprise. These questions fulfill customer needs and align to key responsibilities of the role:

Responsibility	**Question**	**Approaches**
Governance	As teams contribute to the graph, what is their obligation to their downstream consumers?	Schema versioning, deprecation schedules

Responsibility	**Question**	**Approaches**
Governance	Who sets which policies with respect to SLA, SLO, LTS?	RFCs, DevOps discussions, platform policies
Governance	Is deprecation required per-service?	LTS commitments, business product alignment
Governance	Can breaking changes be forced to consumers? Under what circumstances, and on whose accountability?	LTS commitments, business product alignment
Governance	Is support segmented per-platform, in-aggregate, or driven by longest-client-support?	LTS commitments, business product alignment
Advocacy	How do consumers stay informed of changes?	Center of excellence portal, internal communications
Advocacy	How do you ensure clear Graph Policies and usage?	Defined standards, RFCs, templates, and educational programs
Advocacy	How is a new team onboarded successfully?	Center of excellence portal, education
Advocacy	How do we maintain consistency for cross-cutting concerns?	Prioritize RFC and Champion participation, governed consensus
Equip	Which languages, services, and platforms will be supported?	
Equip	How do we create scalable, high-performing teams?	IDE integrations, dev-time tooling, test automation
Equip	How do we automate and enable change in our product(s)?	Schema evolution and registry
Health	Can we automate quality in our delivery?	Tracing-based automated testing, SDLC alignment with GraphQL delivery
Health	Can we observe the health of the graph as a product, not as a series of disjointed services?	Integrated observability, data graph-specific tracing

Education To Support Organizational Change

A comprehensive, continuous education plan has proven crucial to Apollo's customers' success in the enterprise. Once one understands the changes to the organization's graph, a key early step is to educate the teams and management who will drive and support the changes. Graph Champions within the organization have a responsibility to provide education support. Thankfully, both Apollo and the wider GraphQL community have a foundational set of resources.

An example educational outline for GraphQL adoption and change should likely include the following:

- GraphQL introduction:
  - Facebook
  - Reference Implementation
  - Purpose
- Principled GraphQL

Summary

Graph Champions provide essential capabilities to an enterprise's GraphQL consolidation work. When viewed as a product, the value of a data graph serves many technical customers and, ultimately, the business' strategic goals. A successful consolidation strategy needs leaders that can properly equip data graph contributors with the tools they need while also advocating for, governing, and maintaining the overall health of the data graph. Graph Champions are also well-positioned to help an organization navigate some of the challenges of consolidation while providing educational support to graph contributors and consumers alike.

Consolidated Architectures with Federation

By Mandi Wise

Chapter 1 touched on the high-level architecture of GraphQL APIs that are consolidated via federation. By embracing this federated approach, teams can address the lack of consistency that often emerges from other non-federated GraphQL architectures while also exposing data within the graph in a demand-oriented way. In this chapter, we'll explore federation's various implementation details and architectural considerations in greater depth to gain a better understanding of how to fully realize its benefits.

A Better Way to Scale Distributed GraphQL Architectures

The first principle outlined in Principled GraphQL is "One Graph," which states that an organization should have a single unified graph, instead of multiple graphs created by each team. While there are other pathways to a distributed GraphQL architecture, federation is the only option that exposes a single entry point to a data graph while simultaneously allowing teams to maintain logical service boundaries between the portions of the graph that they own and maintain. What's more, federation offers a declarative interface for seamlessly composing the independently managed schemas into a single API, unlike other more brittle, imperative approaches like schema stitching.

We previously discussed that a federated GraphQL architecture consists of two main components: first, a collection of **implementing services**, and second, a **gateway** that sits in front of those services and composes their distinct schemas into a federated data graph. To facilitate schema composition, the gateway and implementing services use spec-compliant features of GraphQL, so any language can implement federation.

Visit the Apollo documentation to view the full federation specification.

Historically at Apollo, we have seen that federation usually isn't a starting point for most enterprises in the early stages of adopting GraphQL. While it can be in some cases, implementing federation before running GraphQL in production with a pre-consolidation pattern will likely necessitate large education and integration efforts for the teams who will be responsible for managing portions of the data graph. It may also skew the focus of this process heavily toward data modelling across services instead of product delivery.

More often, as GraphQL's surface area expands across teams' tech stacks, pain points emerge as these teams attempt to scale within the various pre-consolidation patterns (discussed in Chapter 1) and perhaps even begin to experiment with other non-federated approaches to consolidation. Graph Champions within the organization emerge and drive the teams toward a federated architecture to unify the disparate portions of the data graph, increase developer velocity, and scale GraphQL APIs more effectively.

In our experience, these paths are well-worn and converge on a shift toward a federated data graph. This transition was designed to minimize disruption to teams that are currently contributing to and consuming existing GraphQL APIs. When this transition is properly executed, champions can improve the semantics and expressiveness of the data graph while facilitating improved collaboration between teams. Federated architectures achieve these ends by adhering to two core principles: **incremental adoption** and **separation of concerns**.

Core Principle #1: Incremental Adoption

Just as any GraphQL schema should be built up incrementally and evolved smoothly over time (as outlined in detail as one of the "Agility" principles in Principled GraphQL), a federated GraphQL architecture should be similarly rolled-out through a phased process.

For most teams, a "big bang" rewrite of all existing GraphQL APIs or all portions of a monolithic GraphQL schema may not be fruitful or even advisable. When adopting federation, we recommend that an enterprise identify a small but meaningful piece of their existing GraphQL implementation to isolate as the first implementing service (or a small number of services, if required). Taking an incremental approach to federating the graph will allow you to gradually define services boundaries, identify appropriate connection points between implementing services, and learn as you go.

Additionally, whatever portion of the data graph you scope into an initial implementing service should have at least one client that actively continues to consume this data. From the client's perspective, the transition to federation can and should be as seamless as possible, and continued consumption of this data can help you validate assumptions, test out new federation tooling, and help you understand how to best delineate future implementing services' boundaries.

Core Principle #2: Separation of Concerns

The second core principle of federation is also one of its main architectural advantages when consolidating GraphQL in an enterprise. Federation allows teams to partition the schema of the unified data graph using concern-based separation rather than type-based separation. This distinction sets federation apart from other consolidation approaches like schema stitching and allows teams to collaborate on and contribute to the data graph in a more organic and productive way.

While dividing a GraphQL schema across teams based on types may initially make sense, in practice, types will often contain fields that cannot be neatly encapsulated within a single service's boundaries. For example, where one team maintains a products service and another maintains a reviews services, how do you define the relationship that a list of reviews has to a given product or that a product has to a specific review in these portions of the schema?

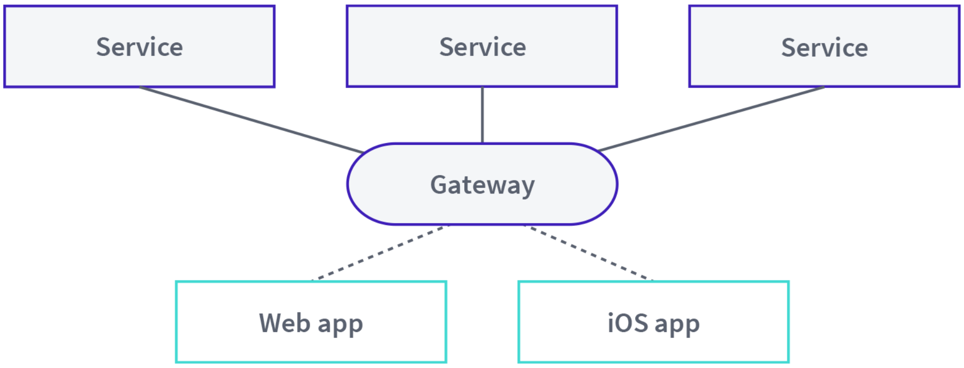
In these instances, foreign key-like fields may find their way into the types, which reduces the expressiveness of relationships between nodes in the graph and exposes underlying implementation details instead of serving product use cases. Alternatively, a non-trivial amount of imperative code would be required to link the types together in a stitched schema.

Concern-based separation allows each service to define the types and fields that it is capable of (and should be responsible for) populating from its back-end data store. The boundaries that encompass these concerns that are related to team structure, geographic hosting, performance, governance and compliance, or some combination thereof. Other services may then directly **reference** and **extend** those types in their schemas with new fields backed by their data stores. Teams maintain their respective portions of the graph with little-to-no friction. The resulting API is a holistic, client-friendly representation of the enterprise's unified data graph.

Apollo Studio provides the necessary tooling to help you in understand references, extensions, and dependencies between graphs. Learn more about Apollo Studio's features.

Implementing Services and the Gateway

To set up a federated data graph, we will need at least one federation-ready implementing service and a gateway GraphQL API to sit in front of it. Note that in practice, a federated data graph will typically have multiple implementing services behind the gateway as follows:



To create an implementing service with Apollo Server, we would also install the `@apollo/federation` package alongside it and use its `buildFederatedSchema` function to decorate the service's schema with the additional federation-specific types and directives. For example:

```
const { ApolloServer } = require("apollo-server");
const { buildFederatedSchema } = require("@apollo/federation");

// ...

const server = new ApolloServer({
  schema: buildFederatedSchema([{ typeDefs, resolvers }])
});

server.listen(4001).then(({ url }) => {
  console.log(`Server ready at ${url}`);
});
```

The `buildFederatedSchema` function ensures that the implementing service's schema conforms to the Apollo Federation specification and also exposes that schema's capabilities to the gateway. In addition to Apollo Server, many third-party libraries provide support for Apollo Federation in a variety of languages including Java, Kotlin, Ruby, and Python.

With an implementing service in place, we can configure a gateway to sit in front of that service. By creating a new Apollo Server in conjunction with the `@apollo/gateway` package, we can **declaratively** compose the implementing service's schema into a federated data graph:

```
const { ApolloGateway } = require("@apollo/gateway");
const { ApolloServer } = require("apollo-server");

const gateway = new ApolloGateway({
  serviceList: [
    { name: "accounts", url: "http://localhost:4001" }
  ]
});

const server = new ApolloServer({
  gateway,
  subscriptions: false,
});

server.listen(4000).then(({ url }) => {
  console.log(`Server ready at ${url}`);
});
```

When the gateway starts up, it uses the URLs provided in the `serviceList` to fetch the schema from each implementing service to compose the federated data graph. In production, we recommend running the gateway in a **managed mode** with Apollo Studio (using static configuration files instead of querying service schemas at start-up), which we'll explore further later in this chapter.

At this time, subscription operations are not supported with Apollo Federation, so the `subscriptions` option must be set to `false`.

The Apollo team has explored other patterns for serving real-time queries with a federated GraphQL API, which you can view in this repository.

When a request reaches the gateway-enabled Apollo Server, it will execute the incoming operation across the implementing services and then form the overall response. How that request is optimized and fulfilled across the federated data graph is determined by a key feature of the gateway known as **query planning**.

At a high level, query planning works by optimizing for the most time spent in a single service to reduce the number of network hops. More specifically, the gateway used a **service-based depth-first approach** to operation execution across services, unlike the breadth-first approach typically used by monolithic GraphQL servers.

Customizing Service-Level Execution

Apollo Gateway also exposes a configuration option called `buildService` that allows both customization of requests before directing them to an implementing service and also modification of responses received from a service before delivering those results to a client. This option can be particularly useful when forwarding auth-related headers from the gateway to the implementing services or when customizing headers sent in a query response.

Connecting the Data Graph with Entities

The core building blocks of a federated data graph are known as **entities**. An entity is a type that we canonically define in one implementing service's schema and then reference and extend by other services. As per the Apollo Federation specification, we define entities in an implementing service's schema using the `@key` directive.

The `@key` directive defines a **primary key** for the entity and its `fields` argument will contain one or more of the type's fields. For example:

```
type User @key(fields: "id") {
  id: ID!
  name: String
  username: String
}
```

The `@key` directive may be used to define multiple primary keys for an entity:

```
type Product @key(fields: "upc") @key(fields: "sku") {
  upc: String!
  sku: String!
  name: String
  price: Int
  brand: Brand
  weight: Int
}
```

The `@key` directive also supports compound primary keys for nested fields:

```graphql
type User @key(fields: "id organization { id }") {
  id: ID!
  name: String
  username: String
  organization: Organization!
}

type Organization {
  id: ID!
}
```

Referencing Entities

After defining an entity in a schema, other implementing services can reference that entity in their schemas. In order for the referencing service's schema to be valid, it must define a stub of the entity in its schema. For example, we can reference a `Product` type defined in one service as the return type corresponding to a `product` field on a `Review` type defined in another service:

```graphql
type Review @key(fields: "id") {
  id: ID!
  body: String
  product: Product
}

extend type Product @key(fields: "upc") {
  upc: String! @external
}
```

Note that the GraphQL spec-compliant `extend` keyword is used before the referenced `Product` type, indicating that this type was defined in another implementing service. The `@key` directive indicates that the reviews service will be able to identify a product by its UPC value and therefore be able to connect to a product based on its `upc` primary key field, but the reviews service does not need to be aware of any other details about a given product. The `@external` directive is required on the `upc` field in the `Product` definition in the review service to indicate that the field originates in another service.

Because the reviews service only knows about a product's UPC, it will be unable to resolve all of a `Product` type's fields. As a result, the reviews service's resolver for the `product` field will only a return a representation of the product with the primary key field value as follows:

```
{
  Review: {
    product(review) {
      return { __typename: "Product", upc: review.upc };
    }
  }
}
```

Resolving References

To resolve any additional fields requested on `Product`, the gateway will pass that representation to the products services to be fully resolved. To fetch the product object that corresponds to the reference, the products service must implement a **reference resolver** for the `Product` type:

```
{
  Product: {
    __resolveReference(reference) {
      return fetchProductByUPC(reference.upc);
    }
  }
}
```

With these resolvers in the place, the gateway can now successfully coordinate execution of operations across service boundaries and clients can make GraphQL query requests to a single endpoint and in a shape that expresses the natural relationship between products and reviews.

Extending Entities

Referencing entities is a key feature of federation, but it's only half of the story. While an entity will be owned by a single implementing service, other services may wish to add additional fields to the entity's type to provide a more holistic representation of the entity in the data graph. Doing so is a simple as adding the additional field to the extended type in a non-originating service. For example, a reviews service's schema may add a `reviews` field to the extended `User` type that was originally defined in an accounts service:

```
extend type User @key(fields: "id") {
  username: String @external
  reviews: [Review]
}
```

The reviews service must then implement a resolver for the user's reviews:

```
{
  User: {
    reviews(user) {
      return fetchReviewsByUsername(user.username);
    }
  }
}
```

When extending entities, it's important to keep in mind that **the entity's originating service will not be aware of the added fields**. Additionally, each field in an entity must only be defined once or the gateway will encounter schema composition errors.

Advanced Extensions, Calculated Fields and Optimizations

Extension points within a data graph can also be leveraged for advanced use cases. In one advanced scenario, an entity may be extended with computed fields by requiring fields from the entity's originating service.

For example, a reviews service could add a custom `reviewName` field for a product by using the `@requires` directive to specify the fields that it depends on from the originating service. Using the `@requires` directives makes these fields available to the reviews service when resolving the `reviewName` field even if they weren't requested by the client in the query operation:

```
extend type Product @key(fields: "sku") {
  sku: String! @external
  name: String @external
  brand: Brand @external
  reviewName(delimeter: String = " - "): String
    @requires(fields: "name brand")
}
```

Multiple implementing services may also resolve a field when data has been denormalized across those services. In this scenario, applying the `@provides` directive on a field definition that returns an extended type will tell the gateway that certain fields for that entity can be resolved by the extending service too:

```
extend type User @key(fields: "id") {
  username: String @external
  reviews: [Review]
}
```

```
type Review @key(fields: "id") {
  id: ID!
  body: String
  author: User @provides(fields: "username")
  product: Product
}
```

The `@provides` directive helps to optimize how data is fetched by potentially eliminating unnecessary calls to additional implementing services. In the above example, the reviews service is capable of resolving an author's username, so a request to the accounts service may be avoided if no additional data is required about the user.

This directive can be a useful (but optional) optimization that helps support the gateway's query planner in determining how to execute a query across as few services as possible, but its usage comes with a few important caveats:

- The implementing service that extends the entity must define a resolver for any field to which it applies the `@provides` directive
- There is no guarantee as to which service will ultimately resolve the field in the query plan
- The `fields` argument of `@provides` does not support compound fields

Extending Query and Mutation Types

As a final note on type extensions, when defining queries and mutations in an implementing service's schema we also add the `extend` keyword in from of the `Query` and `Mutation` types. Because these types will originate at the gateway level of the API, all implementing services should extend these types with any additional operations. For example, `type Query` would be prefixed by the `extend` keyword in the accounts service as follows:

```
extend type Query {
  me: User
}
```

Defining Shared Types and Custom Directives

Value Types

In some instances, implementing services may need to share ownership of a type rather than turning it into an entity and assigning it to a particular service.

As a result, Apollo Federation provides support for shared **value types** including Scalars, Objects, Interfaces, Enums, Unions, and Inputs. When implementing services share value types, then those types must be identical in name in contents, otherwise, composition errors will occur.

> Please see the Apollo Federation documentation for detailed instructions on sharing types across implementing services.

Custom Directives

Apollo Gateway provides support for both type system directives and executable directives. Type system directives are applied directly to an implementing service's schema while executable directives are applied in operations sent from a client.

To provide support for type system directives, Apollo Gateway effectively ignores them by removing all of their definitions and uses from the final composed schema. The definitions and uses of these custom directives remain intact in the implementing service's schema and are processed at that level only.

Executable directives, on the other hand, are treated much like shared value types. These directives must be defined in the schemas of all implementing services with the same locations, arguments, and argument types, or else composition errors will occur. Correspondingly, implementing services should also use the same logic to handling executable directives as well to avoid ambiguity for the clients that apply those directives to operations.

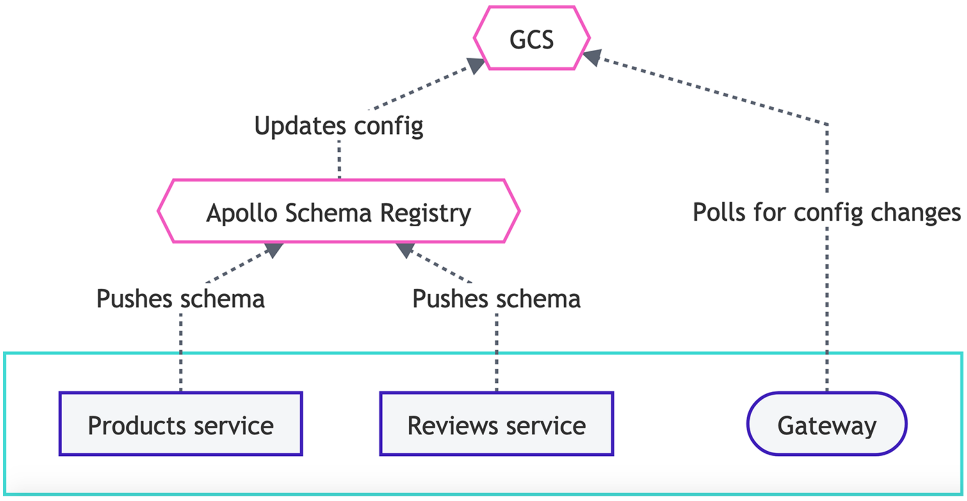
> See the Apollo Federation documentation to read more about handling directives with implementing services.

Managing Cross-Cutting Concerns

Whether sharing value types or executable directives across implementing services, it's always important to consider the long-term implications of introducing cross-cutting concerns that may impede teams' abilities to manage and iterate their portions of the data graph. At Apollo, we've seen enterprises introduce measures into CI/CD pipelines to help manage composition errors as they occur when one team introduces a changes to a shared value type, but be sure to evaluate the complexity that each cross-cutting schema concern adds to your deployment process before doing so.

Managed Federation

In the previous examples, we have seen how to run a federated data graph using a list of service URLs. As a best practice, Apollo Gateway can also run in a managed federation mode and use Apollo Studio as the source of truth for each implementing service's schema. With managed federation, the gateway is no longer responsible for fetching and composing schemas from the implementing services. Instead, each service pushes its schema to a registry, and upon composition, Apollo Studio updates a dedicated configuration file for the graph in Google Cloud Services. The gateway then regularly polls Apollo Studio for updates to the data graph's configuration, as visualized below:



Managed federation supports team collaboration across a distributed GraphQL architecture by allowing each team to safely validate and deploy their portions of the data graph. A managed approach to federation also provides an enterprise with critical observability features to monitor changes in data graph performance via field-level tracing. We will explore managed federation in-depth in relation to graph administration best practices in a later chapter.

Reference Implementation

The majority of the code examples in this chapter were derived from the Acephei example in the Apollo Server repository. You can find the complete source code for the Acephei demo here.

You can also access an example of Acephei's managed, federated data graph in Apollo Studio at demo.apollo.dev.

Summary

In this chapter, we explored the features and benefits of a federated schema and how they may be realized using Apollo libraries. Federation is underpinned by the principles of **incremental adoption** and **separation of concerns**. By adhering to these principles, teams within an enterprise can work toward a consolidated GraphQL architecture along a minimally-disruptive migration path. Federation enables teams to independently, yet collaboratively, manage portions of the single, unified data graph. Entities are the key feature of a federated data graph that provides the extension points among implementing services and power that collaborative work.

With an understanding of the basic mechanics of federation in place, in the next chapter, we'll explore schema design best practices with special consideration for federated data graphs.

Federated Schema Design Best Practices

By Mandi Wise

GraphQL is a relatively new technology, but from its rapid and widespread adoption has emerged a host of common schema design best practices—both from the enterprises that use it at scale every day, as well as the broader developer community. The majority of best practices that apply to non-federated GraphQL schema design also apply when designing service schemas within a federated data graph. However, federated schema design rewards some additional best practices when extracting portions of a data graph into implementing services and determining what extension points to expose between service boundaries.

As we saw in the previous chapter, entities are the core building blocks of a federated data graph, so the adoption of any schema design best practice must be approached with the unique role of entities in mind. A successful federated schema design process should begin by thinking about what the initial entity types will be and how they will be referenced, extended, and leveraged throughout the graph to help preserve the separation of concerns between services—both today and as the graph evolves in the future.

When migrating from a client-only or monolithic GraphQL pattern, that work begins by identifying what entities will be exposed in the first implementing service extracted from the larger schema. When migrating from an architecture consisting of BFF-based GraphQL APIs or any other architecture of multiple overlapping graphs, the work of identifying entities (and determining new service boundaries, in general) may be a bit more complex and involve some degree of negotiation with respect to type ownership, as well as a migration process to help account for any breaking changes that may result for clients.

Whatever your architectural starting point, Apollo Federation was designed to allow the work of identifying entities and defining implementing service boundaries to be done in an incremental, non-disruptive fashion. Beginning to identify these entities is also the essential prerequisite for adopting the other schema design best practices that will follow.

In this chapter, we'll explore some proven best practices for GraphQL schema design with a specific lens on how these practices relate to federated data graphs, as well as any special considerations and trade-offs to keep in mind when designing and evolving schemas across a distributed GraphQL architecture.

Best Practice #1: Design Schemas in a Demand-Oriented, Abstract Way

The shift to a unified data graph is almost invariably motivated in part by a desire to simplify how clients access the data they need from a GraphQL API backed by a distributed service architecture. And while GraphQL offers the promise of taking a client-driven approach to API design and development, it provides no inherent guarantee that any given schema will lend itself to real client use cases.

To best support the client applications that consume data from our federated graph, we must intentionally design schemas in an abstract, demand-oriented way. This concept is formalized as one of the "Agility" principles in Principled GraphQL, stating that a schema should not be tightly coupled to any particular client, nor should it expose implementation details of any particular service.

Prioritize Client Needs, But Not Just One Client's Needs

Creating a schema that is simultaneously demand-oriented while avoiding the over-prioritization of a single client's needs requires some upfront work—specifically, client teams should be consulted early on in the API design process. From a data-graph-as-a-product perspective, this is an essential form of foundational research to ensure the product satisfies user needs. This research should also continue to happen on an ongoing basis as the data graph and client requirements evolve.

Client teams should drive these discussions wherever possible. That means in practice, instead of providing a draft schema to a client team and asking for feedback, it's better to work through exercises where you ask client team members to explain exactly what data is needed to render particular views and have them suggest what the ideal shape of that data would be. It is then the task of the schema designers to aggregate this feedback and reconcile it against the broader product experiences that you want to drive via your data graph.

When thinking about driving product experiences via the data graph, keep in mind that the overall schema of the data graph is a representation of your product and each federated schema is the representation of a domain boundary within the product. This is why Apollo Federation excels at supporting omni-channel product strategies—the data graph can be designed in a demand-oriented way that's based on product functions and the clients that query the graph can, in turn, evolve along with those functions.

Keep Service Implementation Details Out of the Schema

Client team consultation can also help you avoid another schema design pitfall, which is allowing the schema to be unduly influenced by backing services or data sources.

Other approaches to GraphQL consolidation can make it challenging to side-step this concern, but federation allows you to design your schema in a way that expresses the natural relationships between the types in the graph. For example, in a distributed GraphQL architecture without federation, foreign key-like fields may be necessary for an implementing service's schema to join the nodes of your data graph together:

```
type Review {
  id: ID!
  productID: ID
}
```

With federation, however, a reviews service's schema can represent a true subset of the complete data graph:

```
extend type Product @key(fields: "id") {
  id: ID! @external
}

type Review {
  id: ID!
  product: Product
}
```

As another common example of exposed implementation details, here we can see how an underlying REST API data source could influence the names of mutations in a service's schema:

```graphql
extend type Mutation {
  postProduct(name: String!, description: String): Product
  patchProduct(
    id: ID!,
    name: String,
    description: String
  ): Product
}
```

A better approach would look like this:

```graphql
extend type Mutation {
  createProduct(name: String!, description: String): Product
  updateProductName(id: ID!, name: String!): Product
  updateProductDescription(
    id: ID!,
    description: String!
  ): Product
}
```

The revised `Mutation` fields better describe what is happening from a client's perspective and offer a finer-grained approach to handling updates to a product's name and description values where those updates need to be handled independently in a client application. Using two separate update mutations also helps disambiguate what would happen if a client sent the `patchProduct` mutation with no `name` or `description` arguments (because the mutation could handle updating one value or the other, but does not require both for any given operation) and saves the implementing service from having to handle these errors at runtime. We'll speak more on the use cases for finer-grained mutations in the next section.

As a final, related point on hiding implementation details in the schema, we should also avoid exposing fields in a schema that clients don't have any reason to use. If a schema is intentionally and iteratively developed based on the aggregation of product functions and client use cases, then this issue can easily be avoided.

However, when tools are used to auto-generate a GraphQL schema based on backing data sources, then you will almost invariably end up with fields in your schema that clients don't need but may develop unintended use cases for in the future, which will make your schema harder to evolve over the longer term. This is why, at Apollo, we generally discourage the use of schema auto-generation tools—they lead you in precisely the opposite direction of taking a client-first approach to schema design.

Best Practice #2: Prioritize Schema Expressiveness

A good GraphQL schema will convey meaning about the underlying nodes in an enterprise's data graph, as well as the relationships between those nodes. There are multiple dimensions to schema expressiveness—many of which overlap with other schema design best practices—but here we'll focus specifically on standardizing naming and formatting conventions across services, designing purposeful fields in a schema, and augmenting an inherently expressive schema with thorough documentation directly in its SDL to maximize usability.

Standardize Naming and Formatting Conventions

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
>
> — Phil Karlton

Arguably, the "naming things" aspect of this observation grows even more challenging when trying to name things consistently across a distributed GraphQL architecture supported by many teams! (Same goes for caching, but we'll cover that topic separately in a later chapter.)

Being consistent about how you name things may go without saying, but it's even more important when composing schemas from multiple implementing services into a single federated GraphQL API. The "One Graph" principle that drives federation is meant to help improve consistency for clients, and that consistency should include naming conventions. For example, having a `users` query defined in one service and a `getProducts` query defined in another doesn't provide a very consistent or predictable experience for data graph consumers. Similar to fields, type naming and name-spacing conventions should also be standardized across the graph.

Additionally, when an enterprise already has multiple GraphQL APIs in use that will be rolled into the federated data graph, the names of the types within those existing schemas may collide. In these instances, a decision must be made about whether those colliding types should become an entity within the graph or a value type, or if some kind of name-spaced approach is warranted.

The outset of a migration project to a federated data graph is the right time to take stock of what naming conventions are currently used in existing GraphQL schemas within the enterprise, determine what conventions will become standardized, onboard teams to those conventions, and plan for deprecations and rollovers as needed. Additionally, there should also be a thorough review process in place as the graph evolves to ensure that new fields, types, and services adhere to these conventions.

A Brief Note on Pagination Conventions

Another important area of standardization when consolidating GraphQL APIs across an enterprise is providing clients a consistent experience for paginating field results across services. On this topic, we offer these high-level guidelines:

- Add pagination when it's necessary. Don't add pagination arguments to a field when a basic list will suffice.
- When pagination is warranted, leverage your consolidation efforts as an opportunity to standardize type system elements that support pagination (for example, arguments and pagination-related object types and enums).
- Standardizing pagination across your data graph doesn't mean preferring one style of pagination over another (for example, offset-based or cursor-based pagination). Choose the right tool for the job, but ensure that each style of pagination is implemented consistently across services.
- Your internal data graph governance group should actively enforce pagination standards across your implementing services to maintain consistency for clients.

Design Fields Around Specific Use Cases

As mentioned previously, a GraphQL schema should be designed around client use cases, and ideally, the fields that are added to a schema to support those use cases will be single-purpose. In practice, this means having more specific, finer-grained mutations and queries.

While it's still important to ensure that we don't expose unneeded fields in a schema, that doesn't mean we should avoid adding additional queries and mutations to a schema if they are driven by client needs. For example, having two `userById` and `userByUsername` queries may be a better choice than a single `user` query that accepts either a name or ID as a nullable argument. Because the more generalized `user` query could fetch a user by name or ID it necessitates nullable arguments, which creates ambiguity for the client about what will happen if the query is submitted with neither of those arguments included.

Convoluted input types can also complicate the observability story for your data graph. If an input is used to contain query arguments, then each additional field added to the input can make it increasingly opaque as to what field may be the root cause of a particularly slow query when viewing an operation's traces in your observability tools.

Taking a finer-grained approach also applies to update-related mutations. For example, rather than having a single `updateAccount` mutations to rule them all, use more purpose-driven mutations when these values are updated independently by clients. For example, consider this series of mutations used to update a user's account information:

```
type Mutation {
  addSecondaryEmail(email: String!): Void
  changeBillingAddress(address: AddressInput!): Account
  updateFullName(name: String!): Void
}
```

If any of these values needed to be updated simultaneously or not at all, then it would make sense to bundle the updates into a coarser-grained mutation. But with this caveat aside, opting for finer-grained mutations helps avoid the same pitfalls as finer-grained queries do and saves you from doing extra validation work at runtime to determine that the submitted arguments will lead to a logical outcome for a mutation.

As a final note on field use cases, fields within a schema can be leveraged as an entry point to what authenticated users can do within that schema. A common pattern is to add a `viewer` or `me` query to an API, and the GitHub GraphQL API provides a notable example of this pattern:

```
type Query {
  # ...
  "The currently authenticated user."
  viewer: User!
}
```

Document Types, Fields, and Arguments

A well-documented schema isn't just a nicety in GraphQL. The imperative to document the various aspects of a schema is codified in the GraphQL specification. The specification states that documentation is a "first-class feature of GraphQL type systems" and goes further to say that all types, fields, arguments, and other definitions that can be described should include a description unless they are self-descriptive.

So while in many regards a well-designed, expressive schema will be self-documenting, using the SDL-supported description syntax to fully describe how the types, fields, and arguments in an API behave will provide an extra measure of transparency for data graph consumers. For example:

```graphql
extend type Query {
  """
  Fetch a paginated list of products based on a filter.
  """
  products(
    "How many products to retrieve per page."
    first: Int = 5

    "Begin paginating results after a product ID."
    after: Int = 0

    """
    Filter products based on a type.

    Products with any type are returned by default.
    """
    type: ProductType
  ): ProductConnection
}
```

In the example above, we see how a thoroughly described `products` query may look when the query and each of its arguments are documented. And just as with naming conventions, it's important to establish standards for documentation across a federated data graph from its inception to ensure consistency for API consumers. Similarly, there should also be governance measures in place to ensure that documentation standards are adhered to as the schema continues to evolve.

Note that when documenting implementing services' schema files, we can't add descriptions strings above extended types (including extended `Query` and `Mutation` types) because the GraphQL specification states that only type definitions can have descriptions, not type extensions.

Best Practice #3: Make Intentional Choices About Nullability

All fields in GraphQL are nullable by default and it's often best to err on the side of embracing that default behavior as new fields are initially added to a schema. However, where warranted, non-null fields and arguments (denoted with a trailing `!`) are an important mechanism that can help improve the expressiveness and predictability of a schema. Non-null fields can also be a win for clients because they will know exactly where to expect values to be returned when handling query responses. Non-null fields and arguments do, of course, come with trade-offs, and it's important to weigh the implications of each choice you make about nullability for every type, field, and argument in a schema.

Plan for Backward Compatibility

Including non-null fields and arguments in a schema makes that schema harder to evolve where a client expects a previously non-null field's value to be provided in a response. For example, if a non-null `email` field on a `User` type is converted to a nullable field, will the clients that use that field be prepared to handle this potentially null value after the schema is updated? Similarly, if the schema changes in such a way that a client is suddenly expected to send a previously nullable argument with a request, then this may also result in a breaking change.

While it's important to make informed decisions about nullability when initially designing a service's schema, you will inevitably be faced with making a breaking change of this nature as a schema naturally evolves. When this happens, GraphQL observability tools that give you insight into how those fields are used currently in different operations and across different clients. This visibility will help you identify issues proactively and allow you to communicate these changes to impacted clients in advance so they can avoid unexpected errors.

Minimize Nullable Arguments and Input Fields

As mentioned previously, converting a nullable argument or input field for a mutation to non-null may lead to breaking changes for clients. As a result, specifying non-null arguments and input fields on mutations can help you avoid this breaking change scenario in the future. Doing so, however, will typically require that you design finer-grained mutations and avoid using "everything but the kitchen sink" input types as arguments that are filled with nullable fields to account for all possible use cases.

This approach also enhances the overall expressiveness of the schema and provides more transparency in your observability tools about how arguments impact overall performance (this is especially true for queries). What's more, it also shifts the burden away from data graph consumers to guess exactly which fields need to be included in mutation to achieve their desired result.

Tip: Use Default Values for Nullable Arguments and Input Fields

Providing a default value for a nullable argument or input field will also improve the overall expressiveness of a schema by making default behaviors more transparent. In our previous `products` query example, we can improve the `type` argument by adding an `ALL` value to its corresponding `ProductType` enum and setting the default value to `ALL`. As a result, we no longer need to provide specific directions about this behavior in the argument's description string:

```
extend type Query {
  "Fetch a paginated list of products based on a filter."
  products(
    # ...

    "Filter products based on a type."
    type: ProductType = ALL
  ): ProductConnection
}
```

Weigh the Implications of Non-Null Entity References

When adding fields to a schema that are resolved with data from third-party data sources, the conventional advice is to make these fields nullable given the potential for the request to fail or for the data source to make breaking changes without warning. Federated data graphs add an interesting dimension to these considerations given that many of the entities in the graph may be backed by data sources that are not in a given service's immediate control.

The matter of whether you should make referenced entities nullable in an implementing service's schema will depend on your enterprise's existing architecture and likely need to be assessed on a case-by-case basis. Keep in mind the implication that nullability has on error handling—specifically, when a value cannot be resolved for a non-null field, then the null result bubbles up to the nearest nullable parent—and consider whether it's better to have a partial result or no result at all if a request for an entity fails.

Best Practice #4: Use Abstract Type Judiciously

The GraphQL specification currently offers two abstract types in the type system—interfaces and unions. Both interfaces and unions are powerful tools to express relationships between types in a schema. However, when adding interfaces and unions to a schema—and in particular, a federated schema—it's important to do so with a clear-eyed understanding of the longer-term implications of managing these types. To do so, we must first ensure that we're using interfaces and unions in semantically purposeful ways. Second, we must help prepare client developers to handle changes to these types as the schema evolves.

Create Semantically Meaningful Interfaces

A common misuse of interfaces is to use them simply to express a contract for shared fields between types. While this is certainly an aspect of their intended use, they should only be used when you need to return an object or a set of objects from a field *and* those objects may represent a variety of different types with some fields in common. For example:

```
interface Pet {
  breed: String
}

type Cat implements Pet {
  breed: String
  extraversionScore: Int
}

type Dog implements Pet {
  breed: String
  activityLevelScore: Int
}

type Query {
  familyPets: [Pet]
}
```

In this schema, the `familyPets` query returns a list of cats and dogs, with a guarantee that the `breed` field will be implemented on both the `Cat` and `Dog` types. A client can then query for these types' shared fields as usual, or use inline fragments for the `Cat` and `Dog` types to fetch their type-specific fields:

```
query GetFamilyPets {
  familyPets {
    breed
    ... on Cat {
      extraversionScore
    }
    ... on Dog {
      activityLevelScore
    }
  }
}
```

If there was no use case for querying both cats and dogs simultaneously to return both types from a single operation, then the `Pet` interface wouldn't serve any notable purpose in this schema. Instead, it would add overhead to schema maintenance by requiring that the `Cat` and `Dog` types continue to adhere to this interface as they evolve, but with no functional reason as to why they should continue conforming to `Pet`.

What's more, the overhead for maintaining both interface and union types is amplified when dealing with federated data graphs. Where interfaces and unions are shared as value types across schemas, they become cross-cutting concerns (which we'll address further in a later section). Further, interfaces may also be entities in a federated data graph, so challenging decisions may need to be made about which service ultimately "owns" interface entities and whether the services that implement them in a schema can adequately resolve all the types that belong to that interface.

While interfaces are abstract types, they should ultimately represent something concrete about the relationship they codify in a schema and they should indicate some shared behavior among the types that implement them. Satisfying this baseline requirement can help guide your decisions about where to use interfaces selectively in your federated schemas.

Help Clients Prepare for Breaking Changes

Interfaces and unions should be added to a schema and subsequently evolved with careful consideration because subtle breaking changes can occur for the API consumers that rely on them. For example, client applications may not be prepared to handle new types as they are added to interfaces and unions, which may lead to unexpected behavior in existing operations. From our previous example, a new `Goldfish` type may implement the `Pet` interface as follows:

```graphql
type Goldfish implements Pet {
  breed: String
  lifespan: Int
}
```

The previous `GetFamilyPet` query may now return results that include goldfish, but the client's user interface may have been tailored to only handle cats and dogs in the results. And without a new inline fragment in the operation document to handle the `Goldfish` type, there will be no way to retrieve its `lifespan` field value.

As such, it's important to communicate these changes to client developers in advance and it's also incumbent on client developers to treat fields that return abstract types with extra care to guard against potential breaking changes.

Best Practice #5: Leverage SDL and Tooling to Manage Deprecations

Your internal data graph governance group should outline an enterprise-wide field rollover strategy to gracefully handle type and field deprecations throughout the unified graph. We'll discuss graph administration and governance concerns in-depth in the next chapter, so in this section, we'll focus on more tactical considerations when deprecating fields in a GraphQL schema.

GraphQL APIs can be versioned, but at Apollo, we have seen that it is far more common for enterprises to leverage GraphQL's inherently evolutionary nature and iterate their APIs on a rapid and incremental basis. Doing so, however, requires clear communication with API consumers, and especially when field deprecations are required.

Use the @deprecated Type System Directive

As a first step, the `@deprecated` directive, which is defined in the GraphQL specification, should be applied when deprecating fields or enum values in a schema. Its single `reason` argument can also provide the API consumer some direction about what to do instead of using that field or enum value. For instance, in our earlier `products` example we can indicate that a related `topProducts` query has been deprecated as follows:

```graphql
extend type Query {
  """
  Fetch a simple list of products with an offset
```

```
    """
    topProducts(
      "How many products to retrieve per page."
      first: Int = 5
    ): [Product] @deprecated(reason: "Use `products` instead.")

    """
    Fetch a paginated list of products based on a filter type.
    """
    products(
      "How many products to retrieve per page."
      first: Int = 5
      "Begin paginating results after a product ID."
      after: Int = 0
      "Filter products based on a type."
      type: ProductType = LATEST
    ): ProductConnection
  }
```

Use Operation Traces to Assess When It's Safe to Remove Fields

After a service's schema has been updated with new `@deprecated` directives, it's important to communicate the deprecations beyond the SDL as well. Using a dedicated Slack channel or team meetings may serve as appropriate communication channels for such notices, and they should be delivered with any additional migration instructions for client teams.

At this point, a crucial question still remains: "When will it be safe to remove the deprecated field?" To answer this question with certainty that you won't cause any breaking changes to client applications, you must lean on your observability tooling. Specifically, tracing data can provide insight into what clients may still be using the deprecated fields so appropriate follow-ups can be actioned. GraphQL observability tools such as Apollo Studio will check any changes pushed for registered schemas against a recent window of operation tracing data to ensure that a deprecated field rollover can be completed without causing any breaking changes to existing clients.

Best Practice #6: Handle Errors in a Client-Friendly Way

Given that GraphQL offers a demand-oriented approach to building APIs, it's important to take a client-centric approach to handle errors when something goes wrong during operation execution as well. There are currently two main approaches for handling and sending errors to clients that result from GraphQL operations. The first is to take advantage of the error-related behaviors outlined by the GraphQL specification. The second option is to take an “error as data” approach and codify a range of possible response states directly in the schema. Choosing the correct approach for handling a particular error will depend largely on the type of error that was encountered, and, as always, should be informed by real-world client use cases.

Use the Built-in Errors List When Things Really Do Go Wrong

The GraphQL specification outlines certain error handling procedures in responses, so we’ll explore how this default behavior works first. GraphQL has a unique feature in that it allows you to send back both data and errors in the same response (on the `data` and `errors` keys, respectively). According to the GraphQL specification, if errors occur during the execution of a GraphQL operation, then they will be added to the list of errors in the response along with any partial data that may be safely returned.

At a minimum, a single error map in the `errors` list will contain a `message` key with a description of the error, but it may also contain `location` and `path` keys if the error can be attributed to a specific point in the operation document. For example, for the following query operation:

```
query GetUserByLogin {
  user(login: "incorrect_login") {
    name
  }
}
```

The `data` key will contain a `null` user and the `errors` key in the response can be structured with a single error map as follows:

```
{
  "data": {
    "user": null
  },
  "errors": [
    {
      "type": "NOT_FOUND",
      "path": [
        "user"
      ],
      "locations": [
        {
          "line": 7,
          "column": 3
        }
      ],
      "message": "Could not resolve to a User with the login
        of 'incorrect_login'."
    }
  ]
}
```

Many GraphQL servers (including Apollo Server) will provide additional details about errors inside the `extensions` key for each error in the `errors` list. For instance, Apollo Server provides a `stacktrace` key nested inside of the `exception` key of the `extensions` map.

The information inside of `extensions` can be further augmented by Apollo Server by using one of its predefined errors, including `AuthenticationError`, `ForbiddenError`, `UserInputError`, and a generic `ApolloError`. Throwing one of these errors from a resolver function will add a human-readable string to the `code` key in the `extensions` map. For example, an `AuthenticationError` sets the code to `UNAUTHENTICATED`, which can signal to the client that a user needs to re-authenticate:

```
{
  "data": {
    "me": null
  },
  "errors": [
    {
      "extensions": {
        "code": "UNAUTHENTICATED",
        "stacktrace": [...]
      }
    }
  ]
}
```

As a best practice, stack traces should be removed from an error's `extensions` key in production. This can be done by setting the `debug` option to `false` in the Apollo Server constructor, or by setting the `NODE_ENV` environment variable to `production` or `test`.

Please see the Apollo Server documentation for more information on handling, masking, and logging errors in production environments.

The detailed error response that is required by the GraphQL specification and further enhanced by Apollo Server is sufficient to handle any error scenario that arises during operation execution. However, these **top-level errors** that reside in the response's `errors` key are intended for exceptional circumstances and—even with additional, human-readable details in an `extensions` key—may not provide optimal ergonomics for client developers when rendering error-related user interface elements.

For these reasons, the default approach to handling errors is best suited for things that are truly errors. In other words, they should be used when something happened that ordinarily wouldn't happen during the execution of a GraphQL operation. These kinds of errors could include an unavailable service, an exceeded query cost limit, or a syntax error that occurs during development. They are exceptional occurrences outside of the API domain and are typically also outside a client application's end user's control.

Represent Errors as Data to Communicate Other Possible States

Sometimes errors arise during the execution of a GraphQL operation from which a user may recover or reasonably ignore. For example, a new user may trigger a mutation to create a new account but send a username argument that already exists. In other scenarios, certain errors may occur due to situational factors, such as data being unavailable when users are located in some countries.

In these instances, an **errors as data** approach is often preferable to returning top-level errors in a response. Taking this approach means errors are coded directly into the GraphQL schema and information about those errors will be returned under the `data` key instead of pushed onto the `errors` list in the response. As a result, what's returned in the `data` for a GraphQL server response may contain data related to the happy path of an operation or it may contain data related to any number of unhappy path states.

There are different ways to describe these happy and unhappy paths in a schema, but one of the most common is to use unions to represent collections of possible related states that may result from a given operation. Take the following example that includes a `User` type defined in an accounts service and extended to include a `suggestedProducts` field in a products service:

```
# Accounts Service

type User @key(fields: "id") {
  id: ID!
  firstName: String
  lastName: String
  description: String
}
```

```graphql
extend type Query {
  me: User
}
```

```graphql
# Products Service

type Product @key(fields: "sku") {
  sku: String!
  name: String
  price: Float
}

type ProductRemovedError {
  reason: String
  similarProducts: [Product]
}

union ProductResult = Product | ProductRemovedError

extend type User @key(fields: "id") {
  id: ID! @external
  suggestedProducts: [Product]
}

extend type Query {
  products: [Product]
}
```

Above, the `ProductResult` type is a union of the two possible states of a product: it is either available or it has been removed. In the case that a product has been removed, related products can be presented to users in its place. A query for suggested products for a currently logged in user would be structured as follows:

```graphql
query GetSuggestedProductsForUser {
  me {
    suggestedProducts {
      __typename
      ... on Product {
        name
        sku
      }
      ... on ProductRemovedError {
        reason
        similarProducts {
          name
          sku
        }
      }
    }
  }
}
```

Because we are queuing a union type, an inline fragment is used to handle the fields relevant to each union member. The `__typename` field has been added to the operation document to help the client conditionally render elements in the user interface based on the returned type.

Through this example, we can begin to see how errors as data help support data graph consumers in several compelling ways. First, creating a union of happy and unhappy paths provides type safety for these potential states, which in turn makes operation outcomes more predictable for clients and allows you to evolve those states more transparently as a part of the schema.

Second, it also allows you to tailor error data to client use cases. Correspondingly, the requirement to tailor a user experience around error handling is a good indicator that those errors belong in the schema. And conversely, when a data graph is intended to be used predominantly by third parties, it would be impossible to customize error data to suit all possible user interfaces, so top-level errors may be a better option in these instances.

Of course, there's no such thing as an error-handling free lunch. Just as with any union type, clients must be informed of and prepared to handle new result types as they are added to the union (also reinforcing why this approach can be problematic when unknown third parties may query your data graph).

Further, the key to implementing errors as data successfully in a schema is to do so in a way that supports client developers in handling expected errors, rather than overwhelm them with edge-case possibilities or confuse them due to a lack of consistency in adoption across the data graph. An enterprise's data graph governance group must play a key role in setting and enforcing standards for how both top-level and schema-based errors will be handled across teams.

For an in-depth exploration of the errors as data approach, please see the 200 OK! Error Handling in GraphQL talk by Sasha Solomon from GraphQL Summit 2020.

Best Practice #7: Manage Cross-Cutting Concerns Carefully

In the previous chapter, we discussed how sharing value types (scalars, objects, interfaces, enums, unions, and inputs) and executable directives across implementing services' schemas leads to cross-cutting concerns. As a general rule, where implementing services share value types, then those types must be identical in name, contents, and logic, or composition errors will occur. Similarly, executable directives must be defined consistently in the schemas of all implementing services using the same locations, arguments, and argument types, or composition errors will also result.

In some instances, it will make sense for implementing services to share ownership of certain types instead of assigning that type to one service and exposing it as an entity. For example, when a GraphQL API supports Relay-style pagination, it may be necessary to share an identical `PageInfo` object type across multiple services that require these pagination-related fields:

```graphql
type PageInfo {
  endCursor: String
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
  startCursor: String
}
```

It wouldn't make sense to expose `PageInfo` as an entity for several reasons, not the least of which is that there is no obvious primary key that identifies these objects. Further, the fields in this object type will be relatively stable across implementing services and over time, so the likelihood of complications arising from evolving this type is minimal.

There's no simple formula for evaluating the overhead added by a single value type or executable directive in a federated GraphQL API. While they may impact teams' abilities to manage and iterate their portions of the data graph because services may no longer be independently deployable, the long-term cost may be minimal if the types or directives rarely change. As a best practice, your data graph governance group should establish internal guidelines about when to introduce and how to work with value types and executable directives in the data graph, and drive adoption of new measures in your CI/CD pipeline to help manage the composition errors may result from these cross-cutting concerns during deployment.

Summary

In this chapter, we covered a variety of best practices for designing schemas within a federated data graph. We explored what it means to design a schema in a demand-oriented, abstract way with an eye for expressiveness. We also saw how nullability and abstract types can help improve the expressiveness and the usability of a schema when used strategically.

Next, we saw how the `@deprecated` directive and supporting tooling can help teams within an enterprise safely evolve schemas and how using both top-level errors and unions to express a range of possible result states can improve the error handling experience for clients. Finally, we revisited the importance of measuring the cost of adding cross-cutting concerns to a federated data graph.

In the next chapter, we'll move on from focusing exclusively on schema-related concerns to what best practices for overall data graph administration look like in an enterprise.