Federated schema design best practices

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 large, notable companies that use it at scale every day, as well as from 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 graph. However, federated schema design rewards some additional best practices when extracting portions of a graph into subgraphs and determining what extension points to expose between service boundaries.

Throughout this section, we'll explore some proven best practices for GraphQL schema design with a specific lens on how these practices relate to federated 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: Start thinking in entities

Entities are the core building blocks of a federated graph, so the adoption of any schema design best practice must be approached with the unique role of entities in mind. While there's no requirement for subgraphs to define any entities at all with Apollo Federation, the federated schema design process often begins by thinking about what the initial entity types will be and how they will be referenced and extended throughout the graph to help preserve the separation of concerns between services—both today and as the graph evolves in the future.

Define, reference, and extend entities as needed

The Apollo Federation specification indicates that an Object or Interface type can be made into an entity by adding the `@key` directive to its

definition in a subgraph schema. The `@key` directive defines a **primary key** for the entity and its `fields` argument will contain one or more of the type's fields. In the following example, the `Product` entity's primary key would be its `upc` field:

Products Subgraph — Copy

```
type Product @key(fields: "upc") {
  upc: String!
  name: String!
  description: String
}
```

In other words, setting the `upc` field as the key means that other subgraphs that want to use this entity will need to know at least that value for any product. The keys we define should be values that uniquely identify a resource. This is because we want to avoid scenarios where they are used to arbitrarily pass dynamic field data around query execution between subgraphs.

After defining an entity in a schema, other subgraphs 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 the products subgraph as the return type corresponding to a `product` field on a `Review` type defined in reviews subgraph:

Reviews Subgraph — Copy

```
type Review {
  rating: Int
  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 subgraph. 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.

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 subgraph, other services may wish to add additional fields to the entity's type to provide a more holistic representation of the entity in the 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 `Producs` type that was originally defined in the products subgraph:

Reviews Subgraph

Copy

```
type Review {
  rating: Int
  product: Product
}

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

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.

Work from the entities outward

When migrating from a client-only or monolithic GraphQL pattern, that work begins by identifying what entities will be exposed in the first subgraph 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 subgraph 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.

For more information on entity usage and capabilities, please see the Entities documentation page.

Best practice #2: Design schemas in a demand-oriented, abstract way

The shift to a common graph is often 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 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 graph.

When thinking about driving product experiences via the graph, keep in mind that the overall schema of the 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 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 a subgraph's schema to join the nodes of your graph together:

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

Copy

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

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

A better approach would look like this:

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

```
5      id: ID!,
6      description: String!
7    ): Product
8  }
```

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 subgraph 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 #3: Prioritize schema expressiveness

A good GraphQL schema will convey meaning about the underlying nodes in a 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

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 on.)

Being consistent about how you name things may go without saying, but it's even more important when composing schemas from multiple subgraphs 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 graph consumers. Similar to fields, type naming and name-spacing conventions should also be standardized across the graph.

Additionally, when a company already has multiple GraphQL APIs in use that will be rolled into the federated 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 graph is the right time to take stock of what naming conventions are currently used in existing GraphQL schemas within the company, 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 sidebar on pagination conventions

Another important area of standardization when consolidating GraphQL APIs 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 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 company's graph governance group should actively enforce pagination standards across your subgraphs 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 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:

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

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:

```graphql
extend 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 graph consumers. For example:

Copy

```
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 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.

And as a final note, when documenting subgraphs' schema files, we can't add descriptions strings above extended types (including extended `Query` and `Mutation` types in subgraph schemas) because the GraphQL specification states that only type definitions can have descriptions, not type extensions.

Best practice #4: 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 graph consumers to guess exactly which fields need to be included in mutation to achieve their desired result.

And as an additional tip when you do use nullable arguments and input fields, consider providing a default value to 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:

Copy

```
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 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 a subgraph's schema will depend on your existing architecture and internal SLAs and will 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 #5: Use abstract types 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.

As a prerequisite to using an abstract type in a subgraph, that subgraph must be able to resolve all possible concrete types that may be returned by a field that uses the abstract type as its output. Beyond that requirement, we must also ensure that we're using interfaces and unions in semantically purposeful ways. Additionally, 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 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 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:

Copy

```
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 #6: Leverage SDL and tooling to manage deprecations

Your graph governance group should outline a company-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 section, so here 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.

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 #7: 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:

Copy

```
{
  "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:

Copy

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

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.

Lasty, 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.

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 Subgraph

Copy

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

```
  description: String
}

extend type Query {
  me: User
}
```

Products Subgraph

Copy

```
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: [ProductResult]
}

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

```
21 }
```

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:

Copy

```
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 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 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 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 graph. A 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 #8: Manage cross-cutting concerns carefully

Sharing value types (scalars, objects, interfaces, enums, unions, and inputs) and executable directives across subgraphs' schemas leads to cross-cutting concerns. As a general rule, where subgraphs 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 subgraphs using the same locations, arguments, and argument types, or composition errors will also result.

In some instances, it will make sense for subgraphs 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:

Copy

```
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 subgraphs 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 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 graph governance group should establish internal guidelines about when to introduce and how to work with value types and executable directives in the 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

We covered a variety of best practices for designing schemas within a federated graph throughout this section. 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 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 graph.

In the next section, we'll move on from focusing exclusively on schema-related concerns to what best practices for overall graph administration look like for a federated graph.

Edit on GitHub

PREVIOUS
GraphQL consolidation

NEXT
Graph administration