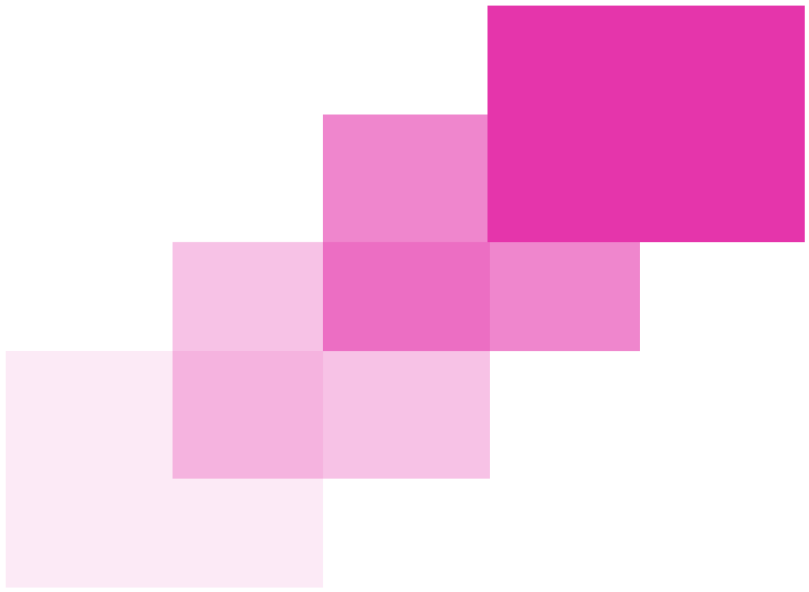


PRODUCTION READY GRAPHQL

Building well designed, performant, and secure GraphQL APIs at scale.

Marc-André Giroux

Table of Contents

Preface

First of all, thanks for purchasing this book. I hope you have as much fun reading it as I had writing it. GraphQL is still a very recent technology but is being adopted more and more every day. I've always thought there wasn't much information available on how to build great GraphQL servers. A lot of the content seems to be focused on GraphQL clients, where GraphQL really shines. For that to happen, we need GraphQL servers to be ready. Over the past 4 years, I've been helping to build GraphQL platforms at Shopify and GitHub, probably two of the biggest public GraphQL APIs out there. This book tries to put into words what I've seen work best and what to avoid.

Enjoy!

Acknowledgments

This book wouldn't have been possible without all the great people I've worked with on GraphQL over the years. Specifically thanks to Robert Mosolgo, Scott Walkinshaw and Evan Huus.

Special thanks to Christian Blais and Christian Joudrey, who gave me a chance at Shopify years ago, and to Kyle Daigle who gave me the opportunity to bring my GraphQL experience at GitHub.

Of course, this book wouldn't have been possible without my wife Gabrielle who has supported me every step of the way.

Illustrations: Gabrielle Le Sauteur Proof reading: Joanne Goulet

An Introduction to GraphQL

Just a few years ago, way before anyone had heard of GraphQL, another API architecture was dominating the field of web APIs: Endpoint based APIs. I call Endpoint-based any APIs based on an architecture that revolves around HTTP endpoints. These may be a JSON API over HTTP, RPC style endpoints, REST, etc.

These APIs had (and still have) several advantages. In fact, they are still dominating the field when it comes to web APIs. There is a reason for this. These endpoints are usually quite simple to implement and can usually answer very appropriately to one use case. With careful design, Endpoint based APIs can very well be optimized for a particular use case. They are easily cacheable, discoverable, and simple to use by clients.

In more recent years, the number of different types of consumers of web APIs has exploded. While web browsers used to be the main clients for Web APIs, we now have to make our APIs to respond to mobile apps, other servers that are part of our distributed architectures, gaming consoles, etc. Even your fridge might be calling a web API when you open the door!

Endpoint based APIs are great when it comes to optimizing an exchange between a client and a server for one functionality or use case. The tricky thing is that because of that explosion in client types, for certain APIs that need to serve that many use cases, building a good endpoint to serve these scenarios became more complex. For example, if you were working on an e-commerce platform and had to provide a use case of fetching products for a product page, you would have to consider web browsers, which may be rendering a detailed view of products, a mobile app which may only display the product images on that page, and your fridge, which may have a very minimal version of the data to avoid sending too much on the wire. What ends up happening in these cases is that we try to build a one-size-fits-all API.

One-Size-Fits-All

What is a One-Size-Fits-All API? It's an API that tries to answer too many use cases. It's an API that started optimized like we wanted and became very generic, due to the failure to adapt to a lot of different ways to consume a common use case. They are hard to manage for API developers because of how coupled they are to different clients, and sometimes of how messy it gets to maintain them on the server.

This became a fairly common problem with endpoint-based APIs, sometimes blamed only on REST APIs. (In reality, REST is not specifically to blame and provides ways to avoid this problem.) Web APIs facing that problem reacted in many different ways. We saw some APIs respond with the simplest solution: adding more endpoints, one endpoint per variation. For example, take an endpoint-based API that provides a way to fetch products, through a `products` resource: `GET /products`

To provide the gaming console version of this use case, certain APIs solved the problem in the following way:

```
GET api/playstation/products

GET api/mobile/products
```

With a sufficiently large web API, you might guess what happened with this approach. The number of endpoints used to answer variations on the same use cases exploded, which made the API extremely hard to reason about for developers, very brittle to changes, and generally a pain to maintain and evolve.

Not everybody chose this approach. Some chose to keep one endpoint per use case, but allow certain query parameters to be used. At the simplest level, this could be a very specific query parameter to select the client version we require:

```
GET api/products?version=gaming

GET api/products?version=mobile
```

Some other approaches were more generic, for example partials:

```
GET api/products?partial=full

GET api/products?partial=minimal
```

And then some others chose a more generic approach, by letting clients select what they wanted back from the server. The JSON:API specification calls them sparse fieldsets:

```
GET api/products?include=author&fields[products]=name,price
```

Some even went as far as creating a query language in a query parameter. Take a look at this example inspired by Google's Drive API:

```
GET api/products?fields=name,photos(title,metadata/height)
```

All the approaches we covered make tradeoffs of their own. Most of these tradeoffs are found between optimization (How optimized for a single use case the endpoint is) and customization (How much can an endpoint adapt to different use cases or variations). Something we'll discuss more deeply during this book.

While most of these approaches can make clients happy, they're not necessarily the best to maintain as an API developer, and usually end up being hard to understand for both client and server developers. Around 2012, different companies were hitting this issue, and lots of them started thinking of ways to make a more customizable API with a great developer experience. Let's explore what they built.

Let's Go Back in Time

Netflix

In 2012, Netflix announced that they had made a complete API redesign. In a blog post about that change, here's the reason they stated:

> Netflix has found substantial limitations in the traditional one-size-fits-all (OSFA) REST API approach. As a result, we have moved to a new, fully customizable API.

Knowing that they had to support more than 800 different devices, and the fallbacks of some of the approaches you've just read, it is not so surprising that they were looking for a better solution to this problem. The post also mentions something crucial to understanding where we come from:

> While effective, the problem with the OSFA approach is that its emphasis is to make it convenient for the API provider, not the API consumer.

Netflix's solution involved a new conceptual layer between the typical client and server layers, where client-specific code is hosted on the server.

While this might sound like just writing many custom endpoints, this architecture makes doing so much more manageable on the server. In their approach, the server code takes care of "gathering content" (fetching data, calling the necessary services) while the adapter layer takes care of formatting this data in the client-specific way. In terms of developer experience, this lets the API team give back some control to client developers, letting them build their client adapters on the server.

Fun fact: They liked their approach so much that they filed a patent for it, with the fairly general name of "Api platform that includes server-executed client-based code".

SoundCloud

Another company struggled with similar concerns back then: SoundCloud. While migrating from a monolithic architecture to a more service-oriented one, they started struggling with their existing API:

> After a while, it started to get problematic, both in regard to the time needed for adding new features, and to the different needs of the platforms. For a mobile API, it's sensible to have a smaller payload footprint and request frequency than a web API, for example. The existing monolith API didn't take this into consideration and was developed by another team, unaware of the mobile needs. So every time the apps needed a new endpoint, first the frontend team needed to convince the backend team that this was truly the case, then a story needed to be written, prioritized, picked, developed and communicated to the frontend team.

Rings a bell doesn't it? This is very similar to the problems Netflix was trying to solve, and the problems that can be caused by implementing the customization solutions we discussed earlier in this chapter.

Their solution to this was quite interesting: instead of including advanced customization options to their main API, they decided that each use case would get its own API server. When you think about it, it makes a lot of sense. This would allow developers to optimize each use case effectively without needing to worry about other use cases, which an endpoint-based API performs really well at.

They called this pattern "Backends for Frontends" (BFF). A great case study from Thoughtworks includes a great visualization of the pattern.

As you can see, this makes each BFF handle one, or many similar experiences, which allows developers to write manageable APIs for one use case and avoids falling in the traps of writing a generic "One-Size-Fits-All" API.

Enter GraphQL

In September 2015, Facebook officially announced the release of GraphQL, whose popularity has since then skyrocketed. However, although it might not be a surprise to you after the other solutions we've covered so far, but it really is in 2012 that Facebook started re-thinking the way they worked with APIs. They were frustrated by very similar concepts:

> We were frustrated with the differences between the data we wanted to use in our apps and the server queries they required. We don't think of data in terms of resource URLs, secondary keys, or join tables; we think about it in terms of a graph of objects and the models we ultimately use in our apps like NSObjects or JSON.

> There was also a considerable amount of code to write on both the server to prepare the data and on the client to parse it.

Unsurprisingly, Facebook was battling with very similar issues. Once again, we see in these quotes the need for a more client-focused, experience-driven APIs. While you read this book, remember these issues, and don't forget GraphQL is simply **one solution**. It is not simply a replacement from HTTP endpoint-based APIs or the "next REST". Instead, think of it as a great solution to building APIs that need to tackle the incredible challenge of building experience based APIs, and redefining the boundary between the client and server, while maintaining sanity on the server-side. With that in mind, let's explore GraphQL!

So what is GraphQL? Maybe a good way to introduce it is by looking at what it is **not**:

- GraphQL is not some sort of graph database
- GraphQL is not a library
- GraphQL is not about graph theory

Here's how I like to describe it instead: **GraphQL is a specification for an API query language and a server engine capable of executing such queries**.

At this point, this may be a bit too vague to truly grasp what GraphQL is all about, but don't worry, let's start with an example. Feel free to jump to the next chapter if you're familiar with GraphQL concepts already.

Hello World

This is the "hello world" of GraphQL. A query that asks for the current user, as well as their name. In this query, `me` and `name` are referred to as *fields*.

```graphql
query {
  me {
     name
  }
}
```

A client sends requests like these to a GraphQL server, usually as a simple string. Here's what the response coming back from a GraphQL server would look like in this case:

```
{
  "data": {
    "me": {
      "name": "Marc"
    }
  }
}
```

Notice how the response and the query are of very similar shapes. A successful GraphQL response always has a `data` key, under which is found the response the client is looking for. GraphQL allows clients to define requirements down to single fields, which allows them to fetch exactly what they need. We can do more than fetching simple fields:

```
query {
  me {
    name
    friends(first: 2) {
      name
      age
    }
  }
}
```

In the above query, we're fetching more than just my name, but we're also fetching the name and age of the first 2 of my friends. As you can see, we're able to traverse complex relationships. We also discover that fields may take arguments. In fact, you can consider fields a bit like functions: they can take arguments and return a certain type. Can you guess what the response would look like?

```
{
  "data": {
    "me": {
      "name": "Marc",
      "friends": [{
        "name": "Robert",
        "age": 30
      }, {
        "name": "Andrew",
        "age": 40
      }]
    }
  }
}
```

The special `query` keyword you see as the start of a query is not a normal field. It tells the GraphQL server that we want to query off the **query root** of the schema. A question you may ask at this point is how did the client even know that this was going to be a valid query to make? Or how did the server express that this was possible?

Type System

At the core of any GraphQL server is a powerful type system that helps to express API capabilities. Practically, the type system of a GraphQL engine is often referred to as the **schema**. A common way of representing a schema is through the GraphQL Schema Definition Language (SDL).

The SDL is the canonical representation of a GraphQL schema and is well defined in the spec. The SDL is such a great tool to express GraphQL schema that we will be using it during the entire book to describe schema examples. The great thing with the SDL is that it is language agnostic. No matter what language you're running a GraphQL API with, the SDL describes the final schema. Here's what it looks like:

```graphql
type Shop {
  name: String!
  # Where the shop is located, null if online only.
  location: Location
  products: [Product!]!
}

type Location {
  address: String
}

type Product {
  name: String!
  price: Price!
}
```

Types & Fields

The most basic and crucial primitive of a GraphQL schema is the Object Type. Object types describe one concept in your GraphQL API. Just by themselves, they're not that useful. What makes them whole is when they define fields. In our previous example, we defined a Shop type that defined three fields: name, location, and products. The `fieldName: Type` syntax allows us to give a return type to our fields. For example, the name field on a Shop type returns a String, the name of the Shop. It is usually helpful to compare GraphQL fields to simple functions. Fields are executed by a GraphQL server and return a value that maps correctly to its return type.

The String type is not user-defined. It is part of GraphQL's pre-defined scalar types. But GraphQL's real power lies in the fact fields, which can return object types of their own:

```
location: Location
```

The location field on the Shop returns a type Location, which is a type that the schema defines. To see what fields are available on a Location type, we must look at the Location type definition:

```graphql
type Location {
  address: String!
}
```

Our Location type defines simply one address field, which returns a String. The fact that fields can return object types of their own is what powers amazing queries like these:

```
query {
  shop(id: 1) {
    location {
      address
    }
  }
}
```

A GraphQL server can execute queries like these because at each level in the query, it is able to validate the client requirements against the defined schema. This visualization might help:

```
query {
  # 1. The shop field returns a `Shop` type.
  shop(id: 1) {
    # 2. field location on the `Shop` type
    # Returns a `Location` type.
    location {
      # 3. field address exists on the `Location` type
      # Returns a String.
      address
    }
  }
}
```

There's still something you might find odd about this query. We know that our `Shop` type has a `location` field, and that our `Location` type has an `address` field. But where is our shop field coming from?

Schema Roots

As you can see, a GraphQL schema can be defined using types and fields to describe its capabilities. However, how can clients even begin to query this "Graph" of capabilities? It must have an entry point somewhere. This is where "Schema Roots" come in. A GraphQL schema must always define a Query Root, a type that defines the entry point to possibilities. Usually, we call that type Query:

```
type Query {
  shop(id: ID): Shop!
}
```

The query type is implicitly queried whenever you request a GraphQL API. Take for example this query:

```
{
  shop(id: 1) {
    name
  }
}
```

The above is a valid GraphQL query because it implicitly asks for the shop field on the Query root, even though we didn't query a particular field that returned a Query type first.

A query root has to be defined on a GraphQL schema, but two other types of roots that can be defined: the Mutation, and the Subscription roots. We'll cover those later in the chapter.

Arguments

You might have noticed something special about the shop field on the Query root.

```
type Query {
  shop(id: ID!): Shop!
}
```

Just like a function, a GraphQL field can define arguments that a GraphQL server can use to affect the runtime resolution of the field. These fields are defined between parentheses after the field name, and you can have as many of them as you like:

```
type Query {
  shop(owner: String!, name: String!, location: Location): Shop!
}
```

Arguments, just like fields, can define a type which can be either a Scalar type or an Input Type. Input types are similar to types, but they are declared in a different way using the input keyword.

```
type Product {
  price(format: PriceFormat): Int!
}

input PriceFormat {
  displayCents: Boolean!
  currency: String!
}
```

Variables

While we are on the subject of arguments, it is good to note that GraphQL queries can also define variables to be used within a query. This allows clients to send variables along with a query and have the GraphQL server execute it, instead of including it directly in the query string itself:

```
query FetchProduct($id: ID!, $format: PriceFormat!) {
  product(id: $id) {
    price(format: $format) {
      name
    }
  }
}
```

Notice that we also gave the query an **operation name**, `FetchProduct`. A client would send this query along with its variables, like this:

```
{
  "id": "abc",
  "format": {
    "displayCents": true,
    "currency": "USD"
  }
}
```

Aliases

While the server dictates the canonical name of fields, clients may want to receive these fields under another name. For this, they can use field **aliases**:

```
query {
  abcProduct: product(id: "abc") {
    name
    price
  }
}
```

In this example, the client requests the `product` field, but defines a `abcProduct` alias. When the client executes this query, it gets back the field as if it was named `abcProduct`:

```
{
  "data": {
    "abcProduct": {
      "name": "T-Shirt",
      "price": 10,
    }
  }
}
```

This comes in useful when requesting the same field multiple times with different arguments.

Mutations

So far, we've covered a lot of GraphQL Schema primitives that allow a client to get information on how to query read-only information from a GraphQL API. At some point, a lot of APIs will want to add functionality that requires writing or modifying data. GraphQL defines the concept of **mutations** to achieve this goal. Similar to the query root we discovered earlier in this chapter, the "entry point" to the mutations of a schema is under the **Mutation*** root. The access to this root in a GraphQL query is done through including the mutation keyword at the top level of a query:

```
mutation {
  addProduct(name: String!, price: Price!) {
    product {
      id
    }
  }
}
```

We can define this `addProduct` mutation in a very similar way we define fields on the query root:

```
type Mutation {
  addProduct(name: String!, price: Price!): AddProductPayload
}

type AddProductPayload {
  product: Product!
}
```

As you can see, mutations are almost exactly like query fields. Two things make them slightly different:

- Top-level fields under the mutation root are allowed to have side effects / make modifications.
- Top-level mutation fields must be executed serially by the server, while other fields could be executed in parallel.

The rest is the same: They also take arguments and have a return type for clients to query after the mutation was made. We'll talk a lot about how to design great mutations in later chapters.

Enums

A GraphQL schema allows us to design our interface in quite a strong way. There are cases where a field can return a set of distinct values, and where our usual scalars would not be the most descriptive to clients. In these cases, it is usually helpful to use Enum types:

```
type Shop {
  # The type of products the shop specializes in
  type: ShopType!
}

enum ShopType {
  APPAREL
  FOOD
  ELECTRONICS
}
```

Enum types allow a schema to clearly define a set of values that may be returned, for fields, or passed, in the case of arguments. They come in very useful to define an API that is easy to use by clients.

Abstract Types

Abstract types are part of a lot of languages, and GraphQL is no different. Abstract types allow clients to expect the return type of a field to act a certain way, without returning an actual type, which we'll usually call the concrete type. There are two main ways to return an abstract type for fields, interfaces, and unions.

Interfaces allow us to define a contract that a concrete type implementing it must answer to. Take this example:

```graphql
interface Discountable {
  priceWithDiscounts: Price!
  priceWithoutDiscounts: Price!
}

type Product implements Discountable {
  name: String!
  priceWithDiscounts: Price!
  priceWithoutDiscounts: Price!
}

type GiftCard implements Discountable {
  code: String!
  priceWithDiscounts: Price!
  priceWithoutDiscounts: Price!
}
```

Here, we have a Product type that implements a Discountable interface. This means the Product type must define the two Discountable fields because by implementing the interface, it must respect that contract.

This allows other fields to return Discountable directly, letting clients know they may request the fields part of that contract directly on the result, without knowing which concrete type will be returned at runtime. For example, we could have a `discountedItems` field which returns a list of either `Product` or `GiftCard` types by directly returning an interface type of `Discountable`.

```graphql
type Cart {
  discountedItems: [Discountable!]!
}
```

Because all types are expected to answer to the Discountable contract, clients can directly ask for the two price fields:

```
query {
  cart {
    discountedItems {
      priceWithDiscounts
      priceWithoutDiscounts
    }
  }
}
```

If a client wants to query the other fields, it must specify which concrete type they want to be selecting against. This is done using fragment spreads or typed fragments:

```
query {
  cart {
    discountedItems {
      priceWithDiscounts
      priceWithoutDiscounts
      ... on Product {
        name
      }
      ... on GiftCard {
        code
      }
    }
  }
}
```

Union types are a bit different. Instead of defining a certain contract, union types are more of a bag of disparate objects that a field could return. They are defined using the union keyword:

```
union CartItem = Product | GiftCard

type Cart {
  items: [CartItem]
}
```

Because it defines no contract and just defines which possible concrete types could be returned by that field, clients have to specify the expected concrete type in all cases:

```
query {
  cart {
    discountedItems {
      ... on Product {
        name
      }
      ... on GiftCard {
        code
      }
    }
  }
}
```

Abstract types are often very useful in a GraphQL schema, but can also be easily abused. We'll cover them in detail in the next chapter.

Fragments

The `... on Product` we were using to select concrete types are called **inline fragments** in GraphQL. Inline fragments are a specific version of a GraphQL query language concept called **fragments**. Fragments allow clients to define parts of a query to be reused elsewhere:

```
query {
  products(first: 100) {
    ...ProductFragment
  }
}

fragment ProductFragment on Product {
  name
  price
  variants
}
```

A fragment is defined using the `fragment` keyword. It takes a **name** and a location where it can be applied, in this case `Product`.

Directives

The last things to cover in the schema definition language are directives. Directives are a kind of annotation that we can use on various GraphQL primitives. The GraphQL specification defines two builtin directives that are really useful: `@skip` and `@include`.

```graphql
query MyQuery($shouldInclude: Boolean) {
  myField @include(if: $shouldInclude)
}
```

In this example, the `@include` directive makes sure the `myField` field is only queried when the variable `shouldInclude` is `true`. Directives provide clients with a way to annotate fields in a way that can modify the execution behavior of a GraphQL server. As you can see above, directives may also accept arguments, just like fields.

It's also possible for us to add custom directives. First, we need to define a name for the directive, and determine where it can be applied.

```graphql
"""
Marks an element of a GraphQL schema as
only available with a feature flag activated
"""
directive @myDirective(
  """
  The identifier of the feature flag that toggles this field.
  """
  flag: String
) on FIELD
```

Then, this directive can be used by clients:

```graphql
query {
  user(id: "1") @myDirective {
    name
  }
}
```

Directives can apply to queries, but they can also be used with the type system directly, making them really useful to annotate our schemas with metadata:

```graphql
"""
Marks an element of a GraphQL schema as
only available with a feature flag activated
"""
directive @featureFlagged(
  """
  The identifier of the feature flag that toggles this field.
  """
  flag: String
) on OBJECT | FIELD_DEFINITION
```

Then, this directive can be applied to schema members directly:

```graphql
type SpecialType @featureFlagged(flag: "secret-flag") {
  secret: String!
}
```

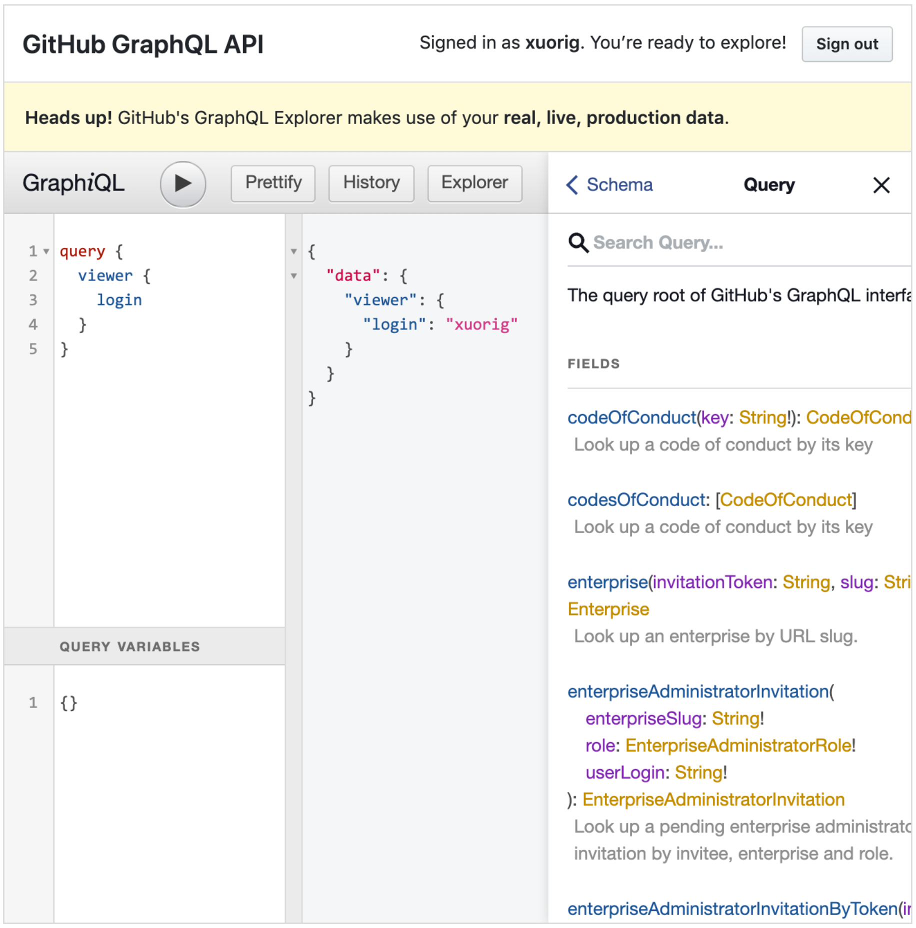
Introspection

The true special power behind GraphQL's type system is its introspection capabilities. With GraphQL, clients can ask a GraphQL schema for what is possible to query. GraphQL schemas include introspection meta fields, which allows clients to fetch almost everything about its type system:

```graphql
query {
  __schema {
    types {
      name
    }
  }
}
```

```
{
  "data": {
    "__schema": {
      "types": [
        {
          "name": "Query"
        },
        {
          "name": "Product"
        },
      ]
    }
  }
}
```

Not only does this help clients discover use cases, but it enables the creation of amazing tooling. For example, GraphiQL, an interactive GraphQL playground, enables users to test GraphQL queries and view the reference for a GraphQL API, thanks to introspection. Here's an example from GitHub's GraphQL developer docs:


GitHub GraphQL API

Signed in as **xuorig**. You're ready to explore! Sign out

Heads up! GitHub's GraphQL Explorer makes use of your **real, live, production data**.

```
query {
  viewer {
    login
  }
}
```

```
{}
```

```
{
  "data": {
    "viewer": {
      "login": "xuorig"
    }
  }
}
```

Search Query...

The query root of GitHub's GraphQL interfa

FIELDS

codeOfConduct(key: String!): CodeOfCond
Look up a code of conduct by its key

codesOfConduct: [CodeOfConduct]
Look up a code of conduct by its key

enterprise(invitationToken: String, slug: Stri
Enterprise
Look up an enterprise by URL slug.

enterpriseAdministratorInvitation(
enterpriseSlug: String!
role: EnterpriseAdministratorRole!
userLogin: String!
): EnterpriseAdministratorInvitation
Look up a pending enterprise administrato
invitation by invitee, enterprise and role.

enterpriseAdministratorInvitationByToken(i

But it doesn't stop there. Introspection is used by clients to generate code and validate queries ahead of time, used by IDEs to validate queries while developing an app, and by so many other tools. Introspection is a big reason why the GraphQL ecosystem is growing so quickly.

Summary

With a query language for clients to express requirements, a type system for servers to express possibilities, and an introspection system that allows clients to discover these possibilities, GraphQL allows API providers to designs schemas that clients will be able to consume in the way they want. These features also support a wonderful ecosystem of tools that we'll cover in this book. Now that we know more about GraphQL and its type system, its time to talk about how to **build** GraphQL servers.

GraphQL Schema Design

So far we've talked a lot about how GraphQL servers express possibilities through their type systems. The key to a great GraphQL API starts with designing these possibilities accurately. Like any API, Library, and even simple functions, GraphQL schema design greatly affects how easy your API will be to use and understand. In this chapter, we'll discuss how to come up with a great design, and also describe a few patterns to avoid or to use when it comes to designing GraphQL APIs specifically.

What Makes an API Great?

Before we even dive into what goes into designing great GraphQL APIs, it's important to discuss what we would like the end result to look like. So what makes a great API in general? Probably my favorite way to but it comes from Joshua Bloch:

> APIs should be easy to use and hard to misuse

A great API should make it easy to do the **right thing** and make it really hard to do the **wrong thing**. Any developer should be able to come to your API and understand with relative ease **what** they can achieve and **how** to do it. Our API should guide developers towards the best way to use the API and also push them away from bad practices, all that **through the way we design it**.

If you're like me you've probably struggled to use some APIs in the past. A lot of us have encountered weird unexpected behaviors, inconsistent wording, out of date documentation, and so many more issues. If you've had the chance to integrate with a well-designed API, then you know how great this can make a developer feel and how quickly it allows any developer to integrate with your offered functionalities. This is why design is the first thing we'll cover in this book: It is such an important part when it comes to building an API that clients will love integrating with.

GraphQL inherently does not make it any easier to design a good API. It comes with a powerful type system, but if we don't use it right, we can fall into the same traps than with any other API style.

Design First

The best way to create a schema that will delight our users is to start thinking of the design very early in the journey. While it is tempting to start implementing right away using your favorite library, taking a design-first approach will almost always result in a better API. Failure to do so will generally lead to a design that is very closely coupled to how things are implemented internally in your system, something your end-users should not have to care about.

You are probably reading this book as a developer interested in building GraphQL APIs. However, if you work for a sufficiently large company, I am fairly certain that you are not the person with the most domain knowledge in all areas your API will cover. This is why it is so important to work with teams and people who are the most familiar with the use cases you are exposing. However, GraphQL does come with some design concerns that we'll cover in this chapter. The best-case scenario is for GraphQL experts and Domain experts to work together.

APIs, especially public ones, are incredibly hard to change once they have been exposed. Thinking of design initially, and making sure we have a great understanding of the concepts enables us to lower the risk of breaking changes down the line.

Client First

As we saw in the introduction, GraphQL is a client-centric API. This philosophy affects how we should design our GraphQL APIs too. While it's tempting to be designing APIs in terms of backend resources or entities, it's very important to design GraphQL APIs with client use cases in mind **first**, before anything else. Doing so ensures we are building an API that answers clients' needs, and is designed in a way that makes it easy to use and hard to misuse as we just saw earlier. Failure to do so often leads to generic APIs that make clients need to guess or read a ton of documentation to achieve what they want, or even worst not being able to achieve it at all.

A great way to ensure you're doing this correctly is working with a "first client" as early on in the process as possible. This means sharing your design early with a client, having them integrate with the API as soon as possible, and even providing mock servers for your API design along the way. We'll talk more about this process in chapter 6. This is especially true for internal and partner APIs, and a bit harder sometimes with public APIs. In any case, you should try to be working with people who are going to be on the receiving end of your API.

Client first does not always mean doing exactly what the clients want either, especially when dealing with a public API. Often, when clients bump into problems, they'll come to you straight with solutions. Gather information on the problem first before implementing the solution they're proposing and take a step back. Chances are there is a more elegant solution behind the problem.

In a similar vein, it turns out that the mantra "You Aren't Going to Need it" (YAGNI), a quote that comes from *ExtremeProgramming* and the agile software community, is particularly useful when it comes to designing APIs at well. Our APIs should be complete, meaning they should provide just enough features for clients to achieve the use cases they're interested in. However, do not expose anything more than that! I can't tell you the number of times I've seen an API needing deprecations or changes because something was exposed without any client in mind, leading to terrible design, or even worst, something that should not be exposed to clients due to performance or security reasons.

Another thing that designing with clients in mind helps with is making sure we are not designing schemas that are influenced by **implementation details**. Developers that will be integrating with your API don't care what database you're using, what programming language the server is built with, or what design issues you have on the backend. Your GraphQL schema is an entry point to **functionality** and you should avoid tying it to any implementation detail on the backend. Of course, concerns like availability and performance will sometimes dictate the design, but it should be done very carefully and with the client in mind. This will not only make for a better API but will also avoid the many problems that come with an external API that is coupled to internal concerns: we want to be able to change internal concerns at a much faster pace than we can with an API that is used by external clients. Keep that in mind when building out your schemas.

GraphQL's typed nature also seems to attract a lot of vendors and tools that offer to build a GraphQL API from your database or other data sources. If you are building an API with a "client first" point of view, **this rarely makes any sense**. This goes against everything we've said so far:

- It makes our schema **coupled to implementation** concerns.
- The result is often very **generic** and in terms of tables and entities.
- It doesn't keep in mind our **clients need** at all.
- It often **exposes way more** than what we should (YAGNI).

I'm not saying these tools are useless; they can be useful for quick prototypes or for when a GraphQL layer on top of a database is actually what is needed. However, for those of us who care about API design and want a curated, cilent-centric and experience-based API, those tools will rarely give the result we're looking for.

The other type of generators is those who will take an existing API definition, Swagger/OpenAPI, and transliterate it into a GraphQL schema. Use these tools with extreme caution. An API should take into consideration the architecture or style it is built in. REST and GraphQL have different design concerns. For example, in a REST API, we should focus on resources and use HTTP methods semantics to interact with them. A very naive generator would generate mutations that look like `postUser` or `putProduct`, when really we are dealing with an API style that should look much more like remote procedure calls with GraphQL: `createUser` and `updateUser` would most likely be the right call in this case. It is possible to tweak these tools to provide a reasonable result, but it will never be as nicely designed as a human-designed, GraphQL-first, curated API. Of course, depending on context this tradeoff might be an acceptable one to make, but it's important to be aware of it.

Now that we've covered what makes a great API, let's on move on to some more specific good practices!

Naming

> The names are the API talking back to you, so listen to them

The quote above is again from **Joshua Bloch**, in How To Design A Good API and Why it Matters. I love this quote and while it mainly relates to designing good Java APIs, it applies very well to web APIs like GraphQL as well.

Naming is very hard to get right but **so important to think about**. A good name immediately conveys information on what the API probably does before even needing to read documentation or even worst, guessing. In fact, naming things in a good way will often guide us towards the right design by itself.

When it comes to naming, consistency is king. Remember that clients have to take the time to understand a new API before using it. When things are consistent, discovering a new API will feel natural to users. Here's a good example:

```
type Query {
  products(ids: [ID!]): [Product!]!
  findPosts(ids: [ID!]): [Post!]!
}

type Mutation {
  addProduct(input: AddProductInput): AddProductPayload
  createPost(input: CreatePostInput): CreatePostPayload
}
```

This schema is not consistent at all when it comes to naming. On the query side, notice how there are two different naming schemes to get a list of objects. On one end, finding posts is prefixed with `find`, and on the other end finding products is done with a simple `products` field. This kind of schema is really hard to use for a client since it's not predictable. A client already using `findPosts` and wishing to integrate with `products` will most likely assume they can do that using `findProducts`, until they're met with a `field does not exist` error or simply not finding what they're looking for. Besides being consistent with action verbs like these, you should also be consistent in how you name your domain concepts as well. For example, don't name a post `BlogPost` somewhere, and `Post` somewhere else unless they are actually different concepts.

Another good example can be found under the `Mutation` type. On one end we can add a product to a shop using `addProduct`, but then adding a social media post is done using `createPost`. These prefix issues are very common. These inconsistencies can add up and make your API really hard to explore and use.

API symmetry is also important. Symmetry could be for example making sure there are symmetric actions possible given a particular entity. For example, a `publishPost` mutation makes it seem like there should be `unpublishPost` mutation. Especially if other mutations in your schema are symmetric in that way. Following the Principle of least astonishment is generally a very good idea.

Another good idea to consider is to be overly **specific** when it comes to naming schema members. Not only does being specific avoid confusion for clients on what exactly that object or field represents, but it helps GraphQL API providers as well. Many large systems have more than one name for similar concepts. When a very common or generic name such as "Event" or "User" is introduced, it often takes up a lot of our naming real estate, making it hard to introduce more specific concepts down the line.

Try to keep the generic names for later, as they might come in useful ahead in the process. Here's an example of naming gone wrong. We introduce a User object which acts as the viewer on the query root. It includes information about the currently logged-in user.

```
type Query {
  viewer: User!
}

type User {
  name: String!
  hasTwoFactorAuthentication: Boolean
  billing: Billing!
}
```

A few months later, the team realizes that this object is being used outside the scope of a logged-in user, for example, when listing team members:

```
type Query {
  viewer: User!
  team(id: ID!): Team
}

type Team {
  members: [User!]!
}

type User {
  name: String!
  hasTwoFactorAuthentication: Boolean
  billing: Billing!
}
```

However, they quickly realize that it makes no sense to expose private information on team members, as they are only meant to be seen by the viewer. They need to either raise an error whenever these fields are accessed by a non-viewer user, or make a new type. The first solution is quite impractical, not great for clients, and pretty brittle as far as implementation goes. The team needs to go through a large deprecation and realizes `User` was more of an interface, while `Viewer` and `TeamMember` should probably have been types of their own:

```graphql
type Query {
  user(id: ID!): User
  viewer: Viewer!
  team(id: ID!): Team
}

type Team {
  members: [User!]!
}

interface User {
  name: String!
}

type TeamMember implements User {
  name: String!
  isAdmin: Boolean!
}

type Viewer implements User {
  name: String!
  hasTwoFactorAuthentication: Boolean
  billing: Billing!
}
```

Specific naming could have avoided a big deprecation and is much nicer to use and understand for clients. Naming things right can guide us towards the right design, and naming things incorrectly just leads us deeper into a bad design.

Descriptions

Most GraphQL schema members may be documented using a description. In terms of SDL, it looks like this:

```graphql
"""
An order represents a `Checkout`
that was paid by a `Customer`
"""
type Order {
  items: [LineItem!]!
}
```

Descriptions are great since they encode this information directly into the schema instead of being found in an external source like documentation. When exploring an API using GraphiQL, users can quickly read descriptions along with the schema.

It's a good idea to describe almost all entities in your schema. Good descriptions should clearly convey what a schema type **represents**, what mutations **do**, etc. However, as much as descriptions are great, they can sometimes reveal inadequate schema design.

Ideally, descriptions are just icing on the cake. A client trying to integrate with your API should rarely **have** to read descriptions to understand how your API can and should be used. As we talked about already, our schema should already convey what things mean and how they should be used. Typical smells are descriptions that describe edge cases, descriptions that contain conditionals and describe contextual behavior.

The bottom line is to use descriptions, but don't make your users rely on them to understand your use cases. We'll talk much more about GraphQL documentation in chapter 10.

Use the Schema, Luke!

GraphQL gives us this amazing and expressive type system to design our APIs. Using it at its full potential makes for incredibly expressive APIs that don't depend on runtime behavior for users to understand. In the next example, we have a schema for a `Product` type which has a `name`, `price`, and `type`. Type is the type of the product like "apparel", "food" or "toys".

```
type Product {
  name: String!
  priceInCents: Int!
  type: String!
}
```

One potential problem with this schema is that clients might have a very hard time trying to understand what can come out of this type. How should they handle the `type` field? If `type` has a set number of items, a much better way to "self-document" this schema, but also provide some runtime guarantees, would be to use an enum type here:

```
enum ProductType {
  APPAREL
  FOOD
  TOYS
}

type Product {
  name: String!
  priceInCents: Int!
  type: ProductType!
}
```

Another common and tempting design issue is having completely unstructured data as part of the schema. This is often done as either a String type with a description that indicates how to parse the field, or sometimes as a scalar type like `JSON`.

```
type Product {
  metaAttributes: JSON!
}

type User {
  # JSON encoded string with a list of user tags
  tags: String!
}
```

A better approach in most cases is to use a stronger schema to express these things.

```
type ProductMetaAttribute {
  key: String!
  value: String!
}

type Product {
  metaAttributes: [ProductMetaAttribute!]!
}
```

While this may look very similar, the typed schema here allows clients to handle this behavior in a much better way, allows us to evolve the schema over time without fearing to break clients, and also lets the server implementation know which fields are used on this `ProductMetaAttribute` type over time. As soon as we use a custom encoding scalar or a string, we lose everything the typed

GraphQL schema gives us. If you really must use it, for example, data structures that are potentially recursive, use it with care. Try to use a strong schema as much as you can.

For example, custom scalars may help turn fields serialized as strings into more useful types:

```graphql
type Product {
  # Instead of a string description, we use a
  # custom scalar to indicate to clients
  # that they can treat the result of this field
  # as valid markdown.
  description: Markdown
}

scalar Markdown
```

And may even be used as input types for more precise validation:

```graphql
input CreateUser {
  email: EmailAddress
}

# A valid email address according to RFC5322
scalar EmailAddress
```

The bottom line is that as soon as we have the opportunity to use a stronger schema, we should consider doing so. This may be through using more complex object types rather than simple scalars, using enum types, and even custom scalars when it makes sense.

Expressive Schemas

GraphQL's type system enables us to build truly expressive APIs. What do we mean by that? An expressive API allows client developers to easily understand how the API is meant to be used. It is easy to use and hard to misuse. Not only that, but as we just saw in the last section, the GraphQL schema enables API providers to express how the API is meant to be used before they even have to look at documentation or worst, having to implement it first.

One way to build expressive schemas is to use nullability to our advantage. Take a look at this example where a GraphQL API is providing a way to find a product. Products can be referred by their global ID or by their name, since we are scoped to a single shop. One way to achieve this is by making a `findProduct` field which optionally accepts both an `id` and `name` field:

```
type Query {
  # Find a query by id or by name. Passing none or both
  # will result in a NotFound error.
  findProduct(id: ID, name: String): Product
}
```

This design would indeed solve the client's needs. However, it is not intuitive at all. What happens if a client provides none of the arguments? What if they provide both? In both of these cases, the server would probably return an error since this is not the way the API is meant to be used. There's a way to solve the same use case, but without having to document it explicitly or letting developers discover this at runtime:

```
type Query {
  productByID(id: ID!): Product
  productByName(name: String!): Product
}
```

This may be a bit surprising to some of you. These fields are so similar; can we not reuse the same field for them? In reality, as we covered a bit in the introduction, we should not be afraid of providing many different ways to do things with GraphQL. Even if we were to expose five different ways of fetching a product, we are not adding overhead to any existing clients. We know that these clients will simply select the ones that answer their use cases best. You can see that the fields now have a single, required field, meaning the API is incredibly hard to misuse. The schema itself, at validation time, will instruct clients how to use the field. Let's look at a more complex example:

```
type Payment {
  creditCardNumber: String
  creditCardExp: String
  giftCardCode: String
}
```

This is a `Payment` type that represents a payment made by a customer. It has three fields that can potentially be filled in, depending on how the order was paid. If a credit card was used, the `creditCardNumber` and `creditCardExp` fields should be filled in. Additionally, if a gift card was used on the order, then the `giftCardCode` field will be present. The first thing we can improve is by using a stronger schema to represent some of these things.

```graphql
type Payment {
  creditCardNumber: CreditCardNumber
  creditCardExpiration: CreditCardExpiration
  giftCardCode: String
}

# Represents a 16 digits credit card number
scalar CreditCardNumber

type CreditCardExpiration {
  isExpired: Boolean!
  month: Int!
  year: Int!
}
```

We've addressed a few issues. Instead of a `String` type credit card expiration, which makes it hard for clients to figure out what format they should provide, we have refactored this into a `CardExpirationDetails` type that has integers fields for both the month and the year. Not only is this more useable than a string clients have to pass, but this also allows us to add fields to this expiration details type as we evolve. For example, we could eventually add an `isExpired: Boolean` field to help clients with this logic. We've also used a custom scalar for the credit card number, which provides a bit more semantics to clients since a credit card number has a pretty particular format.

There is still room for improvement. Notice how all fields are nullable, which means it's quite hard for a client to even know what a payment object will look like at runtime. We can make that better:

```
type Payment {
  creditCard: CreditCard
  giftCardCode: String
}

type CreditCard {
  number: CreditCardNumber!
  expiration: CreditCardExpiration!
}

# Represents a 16 digits credit card number
scalar CreditCardNumber

type CreditCardExpiration {
  isExpired: Boolean!
  month: Int!
  year: Int!
}
```

Much nicer already! We've used an object type that contains all the credit card related input fields. Now, the schema expresses that if a credit card input is passed, then it **must contain all of the fields**, as shown by the fields `number` and `expiration` that are now required. In the previous version, we would have had to handle this kind of conditional in our implementation rather than just letting the schema handle that for us. A common smell that indicates you might need this sort of refactoring is by taking a look at field prefixes. If multiple fields on a type share a prefix, chances are they could be under a new object type. This also lets us evolve the schema in a much better way, instead of adding more fields at the root, in our case the `Payment` object.

Another way to put this principle is the quote "Make Impossible States Impossible". It's hard to find who said it first, but it's a popular saying when it comes to strongly typed languages. What it means is that if we look at our schema's type, this type should make it impossible to have inconsistent information. Let's take a look at a very simple example. We have a Cart object which contains a set of items someone would like to buy. This cart object also has attributes that can let us know if someone has paid for the items, and how much has been paid so far.

```
type Cart {
  paid: Boolean
  amountPaid: Money
  items: [CartItem!]!
}
```

Given this schema, it's unfortunately possible for a client to get data that represents an impossible state. For example, a cart could say it has been paid for, with `amountPaid` being `null`, which makes no sense:

```
{
  "data": {
    "cart": {
      "paid": true,
      "amountPaid": null,
      "item": [...]
    }
  }
}
```

Or the opposite: we could say the cart has not been paid, even though the `amountPaid` is all there:

```
{
  "data": {
    "cart": {
      "paid": false,
      "amountPaid": 10000,
      "item": [...]
    }
  }
}
```

Instead, we want to be building a schema that simply does not allow these states, thanks to the type system. The solution to this always depends on the context. In this case, maybe we were missing concepts. Maybe an `Order` type should represent a paid cart, and a cart simply a bag of objects. In other cases, this can be avoided by simply wrapping related fields under an object type. For example, we could say that if a `payment` property is there on a cart, it means that it has been paid for. We can then use nullability to ensure both paid and amountPaid are present when that's the case:

```graphql
type Cart {
  payment: Payment
  items: [CartItem!]!
}

type Payment {
  paid: Boolean!
  amountPaid: Money!
}
```

Before we move on, here's another example of an API that makes it hard and surprising for a client to use. In this example, we have a `product` field which takes an **optional** sort argument. This could make sense since we don't want to force all clients to pass a value for the sort argument if they're not looking for a particular sort order.

```graphql
type Query {
  products(sort: SortOrder): [Product!]!
}
```

The problem is that the schema tells us absolutely nothing about what the default sort is. Instead, GraphQL provides us with default values, which are incredibly useful to document the default case. So instead of setting a default sort order in our resolving logic, we can encode it right into the schema:

```graphql
type Query {
  products(sort: SortOrder = DESC): [Product!]!
}
```

As you can see, there are many ways to make our schema easier to use and understand:

- Make sure your fields do **one thing well** and avoid clever or generic fields when possible.
- Avoid runtime logic when the schema can enforce it.
- Use complex object and input types to represent coupling between fields and arguments: avoid "impossible states".
- Use default values to indicate what the default behavior is when using optional inputs and arguments.

Specific or Generic

The great debate on specific or generic is one that the API community has been having for many years. On one end, being very specific in the use cases we provide means we optimize very well for the clients interested in that specific functionality. On the other end, an API that is too specific for some clients means it leaves less customization for other clients. This is especially true if you are dealing with many unknown clients since you don't know their use cases just yet.

However, since GraphQL's core philosophy is all about letting clients consume exactly what they need, building our schema with that in mind and opting for simple fields that answer specific client's needs is generally a good idea. Fields that are too generic tend to be optimized for no one and also harder to reason about. Fields should often do one thing, and do it really well. A good indication that a field might be trying to do more than one thing is a boolean argument. You'll often get a better design by splitting it into different fields. Here's a common example:

```
type Query {
  posts(first: Int!, includeArchived: Boolean): [Post!]!
}
```

Instead of making the posts field able to handle both listing archived posts and normal posts, what if we separated it into two fields:

```
type Query {
  posts(first: Int!): [Post!]!
  archivedPosts(first: Int!): [Post!]!
}
```

Just by splitting this into two distinct use cases, we have a more readable schema that is easier to optimize, easier to cache and easier to reason about for clients. However, this is a basic example. The most common examples of excessive use of generic fields are probably very complex filters like these:

```
query {
  posts(where: [
    { attribute: DATE, gt: "2019-08-30" },
    { attribute: TITLE, includes: "GraphQL" }
  ]) {
    id
    title
    date
  }
}
```

While these kinds of filtering syntaxes, which are closer to SQL, are quite powerful, they can be avoided most of the time. Not only are they so generic that they force the server team to handle all the performance edge cases inside a single resolver, but they also don't focus on any use case in particular, making them hard to discover and use on the client too.

Filters can be very useful, for example, if you're implementing an actual search or filtering use case. However, try to be conscious of how much you need these generic fields. Can you expose the specific use case in another way? For example:

```
type Query {
  filterPostsByTitle(
    includingTitle: String!,
    afterDate: DateTime
  ): [Post!]!
}
```

Keep in mind that this whole specific vs. generic debate depends a ton on the kind of API you're building. For example, if you're building an API to support various SQL-like filters for some sort of search interface, going the generic route may make a lot of sense. At this point though, you are just implementing an actual use case!

Anemic GraphQL

Anemic GraphQL is something I stole from the Anemic Domain Model, a pattern popularized by the great Martin Fowler. Anemic GraphQL means designing the schemas purely as dumb bags of data rather than designing them thinking of actions, use cases, or functionality. This is best expressed by an example:

```
type Discount {
  amount: Money!
}

type Product {
  price: Money!
  discounts: [Discount!]!
}
```

Take this `Product` type which represents a product on an e-commerce store. For the sake of this example, the product simply has a price and a list of discounts that are applied at the moment. Now imagine a client wants to display the actual price a customer will have to pay. Pretty simple, right? We take the product's price and remove all discounts off that price.

```
const discountAmount = accounts.reduce((amount, discount) => {
  amount + discount.amount
}, 0);

const totalPrice = product.price - discountAmount
```

The client can happily display this price to customers for the time being. That is until the `Product` type evolves. Imagine that taxes are added to products a few months later:

```
type Product {
  price: Money!
  discounts: [Discount!]!
  taxes: Money!
}
```

Now that taxes are added, the client's initial logic does not hold up anymore, which means customers see the wrong price. What could have prevented that? How about exposing what the client actually was interested in, the total price?

```
type Product {
  price: Money!
  discounts: [Discount!]!
  taxes: Money!
  totalPrice: Money!
}
```

This way, clients consume exactly what they're interested in, and no matter what ends up affecting the `totalPrice`, they will forever get the exact value, avoiding updating their brittle client code every time something is added. This is because we've designed the schema according to our domain, and not simply exposed our **data** for clients to consume.

Let's see how this applies to mutations. Here's a mutation to update a checkout during an e-commerce transaction:

```
type Mutation {
  updateCheckout(
    input: UpdateCheckoutInput
  ): UpdateCheckoutPayload
}

input UpdateCheckoutInput {
  email: Email
  address: Address
  items: [ItemInput!]
  creditCard: CreditCard
  billingAddress: Address
}
```

At first glance, this seems great for clients. It lets them modify any attribute of that checkout. When we look closer, we realize that there are a few problems with this approach:

- Because the mutation focuses so much on data, and not on behaviors, our clients need to guess how to make a specific action. What if adding an item to our checkout actually requires updates to a few other attributes? Our client would only learn that through errors at runtime, or worst, may end up in a wrong state by forgetting to update one attribute.
- We've added cognitive overload to clients because they need to select the set of fields to update when wanting to take a certain action, like "Add to Cart".
- Because we focus on the shape of the internal data of a Checkout, and not on the potential behaviors of a Checkout, we don't explicitly indicate that it's even possible to do these actions, we let them guess by looking at our data model.
- We've had to make everything nullable, which makes the schema much less expressive in general.

An alternative to these coarse-grained generic mutations is to go the fine-grained approach:

```graphql
type Mutation {
  addItemToCheckout(
    input: AddItemToCheckoutInput
  ): AddItemToCheckoutPayload
}

input AddItemToCheckoutInput {
  checkoutID: ID!
  item: ItemInput!
}
```

We've addressed a lot of the issues we just outlined:

- Our schema is strongly typed. Nothing is optional in this mutation. Our clients know exactly what to provide to add an item to a checkout.
- No more guessing. Instead of finding which data to update, we add an item. Our clients don't care about which data needs to be updated in these cases, they just want to add an item.
- The set of potential errors that may happen during the execution of this mutation has been greatly reduced. Our resolver can return finer-grained errors.
- There's no way for the client to get into a weird state by using this mutation because that's handled by our resolver.

An interesting side effect of this design is that the server-side implementation of such mutations is generally much nicer to both understand and write because it focuses on one thing and because the input and payload are predictive. In a world with pub/sub subscriptions, it's also much easier to see which events should be triggered by certain mutations.

The Relay Specification

Before we dive into other schema design concepts, we must talk about the Relay Specification. Relay is a JavaScript client for GraphQL that was initially released by Facebook along with GraphQL. Relay is a powerful abstraction for building client applications consuming GraphQL. However, to achieve this, it makes several assumptions about the GraphQL API it interacts with:

- A method to re-fetch objects using a global identifier
- A `Connection` concept that helps paginate through datasets.
- A specific structure for mutations

We'll cover these points in more detail when we talk about their design implications. But for now, just remember that when we talk about Relay, we talk about these specific assumptions / design considerations that the **Relay client** expects.

Lists & Pagination

Almost all schemas will eventually expose list type fields. Almost all simple examples we've seen so far exposed these things as simple list types:

```
type Product {
  variants: [ProductVariant!]!
}
```

This is the simplest way to allow clients to fetch lists but can be a terrible decision down the line. In this example, the `variants` field provides no control to clients into how many items will be returned. This means that no matter how many variants there are for a certain product, they will be **all** returned to the client at run time. This often leads to fields that have to be removed due to performance issues on the backend, or clients needing to do their own filtering (wasting huge amounts of data) since they only want to display the first few items in their UI for example.

For this reason, **pagination** is almost always an essential component of a good API. The idea behind it is to break up large datasets into "pages", letting the client get parts of that data instead of sending way too much data across the wire. Pagination makes for a great experience for both clients and servers:

- On the server-side, pagination helps us load a certain part of a dataset, instead of making queries for way too much data, which will often lead to extremely slow request times and timeouts over time.
- On the client-side, this often encourages a better user experience and performance as well. We would not want our users to have to scroll through a list of thousands of items.

Practically, pagination is done in various ways in a GraphQL schema. Let's cover the two main approaches.

Offset Pagination

The most widely used technique for paginating an API is what's called offset pagination. In offset pagination, the clients tell us how many items they are interested in receiving, but also an `offset` or `page` parameter that helps them move across the paginated list. In an HTTP API, this often would look like this:

`GET /products?limit=250&page=3`

With GraphQL, we could recreate this behavior on fields:

```
type Query {
  products(limit: Int!, page: Int!): [Product!]!
}
```

This type of pagination can be great since it's often the easiest way to implement it on the backend, but it also gives a lot of flexibility to the client since they can skip to any page they're interested in and track their location as they're paginating through the list. But while it's effective, as API providers grow, a lot of them start noticing some issues with the technique. The first reason is often database performance. If we take the `products` field we designed and try to imagine what a database query would look like to fulfill it, it would usually look a bit like this:

```
SELECT * FROM products
WHERE user_id = %user_id
LIMIT 250 OFFSET 500;
```

These types of SQL queries often don't scale well for very big datasets. When the `offset` parameter grows large, the database implementation often needs to read all rows up to this number just to get to that offset, but then has no need for all the rows it just read, and simply returns `offset + limit` to the user.

The second common problem with offset pagination is that they can return inconsistent results. This is because any changes to the list while a client is paginating can modify the items returned in the page requested by the client. Imagine a client fetching the first page, while items are getting added to the list. When the client loads the second page, it's now possible it will receive some duplicate results from the first page since they were "pushed down" by new results. There are ways around this problem, but still something that needs to be addressed.

For these two reasons, a lot of API providers find themselves moving to another style of pagination that is **cursor-based**.

Cursor Pagination

Cursor pagination approaches the problem from a different perspective. A **cursor** is a stable identifier that points to an item on the list. Clients can then use this cursor to instruct the API to give them a number of results before or after this specific cursor. In practice, this looks like this:

```
type Query {
  products(limit: Int!, after: String): [Product!]!
}
```

Notice how the concept of "pages" does not exist in cursor pagination. The only information a client knows is the "next" or "previous" few items, but generally does not know how many pages there are, and can't skip ahead to any page. The performance downsides of offset pagination are eliminated because we can now use that cursor to fetch the results:

```
SELECT * FROM products
WHERE user_id = %user_id
AND id >= 15
ORDER BY id DESC
LIMIT 10
```

In cursor pagination, the server always provides what the "next" cursor is, in some way or another, to allow clients to request the next few elements. For example:

```
{
  "data": {
    "products": {
      "next": "def456",
      "items": [{},{},{}]
    }
  }
}
```

A client would then take the next cursor and use it to fetch the next 10 items:

```
query {
  products(first: 10, after: "def456) {
    next
    items {
      name
      price
    }
  }
}
```

Unless you absolutely need the ability for clients to skip ahead to different pages, cursor pagination is a good choice for GraphQL APIs. Today, most GraphQL APIs use cursor-based pagination, and that is mostly due to Relay's connection pattern.

Relay Connections

As covered earlier, Relay has certain assumptions about how a GraphQL schema should be designed. One of those assumptions is about how a GraphQL API should handle pagination. Unsurprisingly, it is strongly based on cursor pagination, but the way it is designed is very interesting. Relay calls paginated lists

Connections. To better understand the connection abstraction, let's start by what it looks like for a client to query a connection field:

```
query {
  products(first: 10, after: "abc123") {
    edges {
      cursor
      node {
        name
      }
    }
    pageInfo {
      endCursor
      hasNextPage
      hasPreviousPage
    }
  }
}
```

There's a lot to unpack here. Connections return a **connection type** that exposes two fields. The `edges` field contains the data we requested, while `pageInfo` is a field that contains metadata about the pagination itself. The `edges` field does not return the items immediately but instead returns an edge type with extra connection metadata for that particular item. This is how we know the cursor of each item in the list. Finally, the node field on the list of edges is what contains the data we were looking for, in our case the products. The result for such a query would look something like this:

```
{
  "data": {
    "products": {
      "edges": [
        {
          "cursor": "Y3Vyc29yOnYyOpHOAA28Nw==",
          "node": {
            "name": "Production Ready GraphQL Book"
          }
        }
      ],
      "pageInfo": {
        "endCursor": "Y3Vyc29yOnYyOpHOAA28Nw==",
        "hasNextPage": true,
        "hasPreviousPage": false
      }
    }
  }
}
```

A client would then take the `endCursor` in the `pageInfo` metadata, and use it to get the next items after what we've already fetched. The full schema for a connection usually looks like this:

```
type ProductConnection {
  edges: [ProductEdge]
  pageInfo: PageInfo!
}

type ProductEdge {
  cursor: String!
  node: Product!
}

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

type Product {
  name: String!
}
```

At first, the connection pattern appears to be overly verbose. After all, what most clients are interested in is getting a list of products, why wrap it into this weird "edges" concept? It turns out that while the connection pattern comes with a certain overhead to clients, it is incredibly useful to design more complex scenarios. For example, the GitHub API uses connection edges to encode data about the **relationship** rather than the item itself. For example, the `Team.members` connection and edges have the role of a user in that team of the edge type rather than on the User type itself:

```
type TeamMemberEdge {
  cursor: String!
  node: User!
  role: TeamMemberRole!
}
```

The connection pattern as defined by Relay does require cursor-based pagination, but note that the edge types are still a good idea, even if you opt for offset based pagination. Using the underlying type directly in a list type is rarely what we want, as seen in the example above, where some fields are specific to a type's membership in a collection.

Custom Connection Fields

There are a few fields that providers often provide along with the connection and edge types. Clients often find the `edges { node }` syntax to be overly verbose. One thing I've seen certain providers offer is a way to skip the `edges` part and get all nodes directly. We still get the benefits of paginated lists but without the verbosity of the `edges` pattern:

```
query {
  products(first: 100) {
    nodes {
      name
    }
  }
}
```

Remember to provide **both options** if you choose to implement this helper field. As we saw, the edge pattern gives us a ton of advantages when it comes to designing relationships.

The next field that is commonly added to connections is a `totalCount` field, which allows clients to request the total amount of nodes (before pagination) found in the list. I'm hesitant about recommending this one as a good practice. Don't add this one by default on all connections. Computing a `totalCount` on large collections can often be very slow, depending on where the collection is resolved from. There's no denying the field can come in useful in certain cases but performance should be kept in mind before exposing it. Once it's there, it's very hard to remove or maintain with performance issues.

Pagination Summary

These very common page number links that are used for pagination are very easy to implement with offset-based pagination, but almost impossible to do well with a cursor-based pagination, which is more geared towards "infinite list" use cases. If this is something you absolutely have to support, go for offset pagination, but remember the possible drawbacks in the future. I highly encourage to use a "Connection style" pattern even with offset pagination to allow you to represent relationships in a better way than simple list types.

If your use case can be supported by cursor-based pagination, I highly recommend choosing the connection pattern when designing your GraphQL API:

- Cursor pagination is generally a great choice for accuracy and performance.
- It lets Relay clients seamlessly integrate with your API.
- It is probably the most common pattern in GraphQL at the moment and lets us be consistent with other APIs in the space.
- The connection pattern lets us design more complex use cases, thanks to the Connection and Edge types.

Sharing Types

As your schema grows, an appealing thing will be to reuse types across different fields and use cases. Sometimes, it makes total sense. However, trying to share **too much** rarely turns out well. It might be the most common problem I've seen in GraphQL schemas. While the appeal of reusing types (for client-side structure reuse for example) can be great, it can lead to a lot of pain down the line. A good example is related to the connection pattern we just talked about. Imagine this schema where an `Organization` has a paginated connection of `users`:

```graphql
type UserConnection {
  edges: [UserEdge!]!
  pageInfo: PageInfo!
}

type UserEdge {
  node: User
}

type User {
  login: String!
}

type Organization {
  users: UserConnection!
}
```

Now imagine we add the concept of teams, where teams have members. A common mistake would be to reuse the same `UserConnection` for the `members` field. After all, team members and organization users are all users, right?

```graphql
type UserConnection {
  edges: [UserEdge!]!
  pageInfo: PageInfo!
}

type UserEdge {
  node: User
}

type User {
  login: String!
}

type Organization {
  users: UserConnection!
  teams: [Team]
}

type Team {
  members: UserConnection!
}
```

The issue with reusing types like this is that the future holds many surprises with how types diverge over time. This example is something I've seen multiple times. Once we realize that team members have different features than organization users, we get stuck:

```graphql
type UserEdge {
  isTeamLeader: Boolean
  isOrganizationAdmin: Boolean
}
```

Our `UserEdge`, where we would have loved to put information related to a User within a team, is shared across both organization users and team members, which means we can't add anything specific to one of them. Now imagine if we had `TeamMemberConnection` and `OrganizationUserConnection`. We would be free to encode any data on the edges and connections. This is a really good example of the dangers of sharing types.

Another common approach is trying to share inputs. For example, `createX` and `updateX` mutations often look fairly similar. The update mutation will often take the id of the resource to update, but the rest of the attributes will be incredibly similar to the create one. An approach that we can use is sharing an input between both:

```graphql
input ProductInput {
  name: String
  price: MoneyInput
}

type Mutation {
  createProduct(input: ProductInput):
    CreateProductPayload

  updateProduct(id: ID!, input: ProductInput):
    UpdateProductPayload
}
```

This approach can be useful to allow generated clients to reuse forms and logic, but it can also lead to a problem similar to the connection example we saw. A create input would usually have a bit more non-null fields, because you can't create a product without a name, for example. Because we reuse it in the update mutation, we have to make that input field nullable, meaning the create mutation must handle this validation at runtime instead of simply letting the schema do it. We'll see more about mutation design later on in this chapter.

The two examples we covered here are the reasons why I generally recommend against trying too hard to share types unless it's very obvious it can be shared. When there are any doubts, the downsides usually outweigh the benefits.

Global Identification

Another concept that gained popularity in GraphQL APIs is global identification of objects. Once again this originally comes from Relay, but has since become a good practice in general. The idea is that a GraphQL client should be able to fetch any "node" in the graph given a unique identifier. In practice, this translates in GraphQL server exposing a global `node(id: ID!): Node` field that lets clients fetch any node through that single field. The node field returns a `Node` interface type:

```
interface Node {
  id: ID!
}

type User implements Node {
  id: ID!
  name: String!
}
```

This Node interface is a means of saying this object has a globally unique ID and may be fetched using `node(id: ID!)` and `nodes(ids: [ID!]!)` fields. What's the goal of all this? A lot of it is client-side caching. GraphQL clients often build complex normalized caches to store nodes they have previously fetched. Relay needs a mechanism to re-fetch single nodes, and having the node convention allows the client to do it without too much configuration. Do you absolutely need global identification? Not necessarily. If you don't expect any Relay clients to use your API and don't see a need to fetch objects using a single global identifier, you don't have to implement it. For example, clients like Apollo can use a combination of the type name and a simpler ID to build that global identifier for you.

Nonetheless, having a globally unique ID for a certain "node" or "object" can be quite a useful principle. In fact, they are similar to uniform resource identifiers (URIs) used to identify REST resources for example. One important part of these identifiers, just like REST URIs, is that users should not try to build or hack their IDs, but instead simply use the IDs they get from the API directly. A good way to make sure this happens is by using **opaque identifiers**:

```
{
  "data": {
    "node": {
      "id": "RmFjdGlvbjoy"
    }
  }
}
```

The most common way to make the IDs opaque is to `Base64` them. The goal here is not to completely hide how the ID is built (It's easy for a client to see it is Base64). Instead, this simply reminds the clients that the string is meant to be opaque. Opaque IDs are great since they enable us to change the underlying ID generation knowing that clients have hopefully not built logic coupled to how the ID is constructed. However, sometimes, I find that they don't always lead to the best developer experience. It can be hard to know what kind of node ID you have in your hands when building a client application. An idea I'm interested in is opaque ids that still let you know a bit about themselves. I believe the first time I encountered these were with the Slack API. They had opaque tokens that started with a different letter based on the object's type. Maybe a good idea for global IDs is to include a bit of information to help developers.

As far as building these IDs, you should include as much information as possible that would help fetch this node globally. In the most basic cases, this is usually `type_name:database_id`, but you should not always default to this. Sometimes nodes cannot be fetched without routing information, especially in more distributed architectures. Make sure you include any information that would make it easier to "route" to this node. For example, "products" might be shared or distributed by "shops", meaning our IDs might need the shop id in there: `shop_id:type_name:id`.

To summarize:

- You don't always need global identification, especially if not planning on supporting Relay.
- Like Connections, they can be a good pattern even outside of a Relay context.
- Opaque IDs are recommended.
- Ensure your global ID contains enough context to globally route to a node, especially in a distributed architecture.

Nullability

Nullability is a GraphQL concept that allows us to define whether or not a field can return `null` or not when queried. As we saw when introducing GraphQL, a non-null field, meaning a field that cannot return null at runtime is defined by using the bang (!) symbol after its type. By default, all fields are nullable:

```
type Product {
  # This field is non-null
  name: String!

  # Price returns null when the product is free (default)
  price: Money

  # The tags field itself can be null.
  # If it does return a list, then every
  # item within this list is non-null.
  tags: [Tag!]
}
```

If a field returns null when queried, even though it was marked as non-null in the schema, this causes a GraphQL server to error. Since the field can't be null, a GraphQL will go up the parent of the field until it finds a nullable one. If fields were all non-nullable, the entire query returns `null`, with an error. For example, let's say that in the following query, the `topProduct` field is marked as non-null, and the `shop` field was marked as nullable.

```
query {
  shop(id: 1) {
    name
    topProduct {
      name
      price
      tags
    }
  }
}
```

Now imagine that the name field was to return `null`, even though we marked it as non-null. Because `topProduct` can't be null either, the result we would get would look like this:

```
{
  "data": {
    "shop": null
  }
}
```

Because `shop` was nullable, GraphQL had to bomb the entire shop type response even though only a single field `name` ended up returning `null`. This example is a good reminder that nullability can be either a really powerful thing or a terrible mistake, depending on how we apply it.

Non-Nullability is great for many reasons:

- As covered earlier in the chapter, it helps to build more expressive and predictable schemas.
- It lets clients avoid overly defensive code/conditionals.

However, it does come with several dangers to keep in mind:

- Non-null fields and arguments are harder to evolve. Going from non-null to null is a breaking change, but the opposite is not.
- It's very hard to predict what can be null or not, especially in distributed environments. Chances are your architecture will evolve, and anything including timeouts, transient errors, or rate limits could return null for certain fields.

So here are a few guidelines that I use for nullability when designing schemas:

- For arguments, non-null is almost always better to allow for a more predictable and easy to understand API. (If you're adding an argument, nullable is the best choice to avoid breaking existing clients)
- Fields that return object types that are backed by database associations, network calls, or anything else that could potentially fail one day should almost always be nullable.
- Simple scalars on an object that you know will already have been loaded on the parent at execution time are generally safe to make non-null.
- Rarely: For object types you're strongly confident will never be null, and that will still allow for partial responses at the parent level. (Hard to predict, especially when the type ends up being reused in other contexts)

Following these guidelines will usually lead you towards the right decision on whether to make a schema member non-nullable or nullable. As always, this depends a lot on implementation, so when in doubt, make sure you understand the underlying system before deciding on nullability!

Abstract Types

Abstract types are incredibly useful when designing a GraphQL schema. They can be of great help when designing a schema and can be truly helpful to decouple our interface from the underlying persistence layer. For example, take a look at this type that was generated from our database model:

```
type SocialMediaFeedCard {
  id: ID!
  title: String!
  birthdayDate: DateTime
  eventDate: DateTime
  postContent: String
}
```

What this type aims to represent is a social media post. The problem is that this post is sometimes about a birthday, sometimes about an event, and sometimes simply a text post. As you can see, because of the way we've designed this, we don't use the schema to its full potential and it's possible to be in illegal states. A birthday card should not have content, simply a `birthdayDate`. An event should not also include a `birthdayDate`, yet our schema implies this is possible.

We can instead design this concept using abstract types, in this case by using an Interface type:

```
interface SocialMediaFeedCard {
  id: ID!
  title: String!
}

type BirthdayFeedCard implements SocialMediaFeedCard {
  id: ID!
  title: String!
  date: DateTime!
}

type EventFeedCard implements SocialMediaFeedCard {
  id: ID!
  title: String!
  date: DateTime!
}

type ContentFeedCard implements SocialMediaFeedCard {
  id: ID!
  title: String!
  content: String!
}
```

The schema is instantly clearer for clients, and we easily see the possible card types that could be coming our way. We also don't need any nullable fields anymore and all the potential types don't allow for any illegal states as we had before. That is the power of abstract types.

Union or Interface?

GraphQL has two kinds of abstract types: Union types and Interface types. When should you use which? A good rule of thumb is that interfaces should be providing a common contract for things that share **behaviors**. Take GitHub for example, which has a `Starrable` interface for objects that can be "starred": Repositories, Gists, Topics. Unions should be used when a certain field could return different types, but these types don't necessarily share any common behaviors. A common example of this is for a `search` field that returns a list type which can contain many different possible object types, but not necessarily share common behaviors.

Don't Overuse Interfaces

Interfaces are great to create some stronger contracts in our schema but they are sometimes over-relied on, for example when they are simply used to share common fields. When multiple types share a couple of fields but these types don't share any common behavior, try to avoid the temptation to just throw an interface in the mix. A good interface should mean something to the API consumers. It describes and provides a common way to do or behave like something instead of being or having something.

So how do we know if our interfaces are too focused on "categorizing" objects, grouping similar attributes, and not focused enough on interactions and behaviors? One useful schema "smell" is **naming**. When we use interfaces that don't have strong meaning in the schema, the naming will usually be awkward and meaningless. A common example of this is having the word Interface in the type name, for example, `ItemInterface`, which would share common fields with an item in an e-commerce domain. As our Cart items, a Checkout items, an Order items start drifting apart, this interface might become a pain to maintain since it looks like it was designed purely in terms of what common data is shared. If we had looked at all sorts of items and determined they had very different interactions and behaviors inside our domain, maybe we would have picked a different abstraction. I've seen other namings such as `ItemFields` or `ItemInfo` used in the same way.

The most common way I see interfaces used for the wrong reason is when the GraphQL implementation used makes it easy to use interfaces for **code reuse**. Instead of using the GraphQL schema to allow for code reuse, use the tools you have in your programming language to make it easier to reuse fields. For example, use helper functions, composition, or inheritance.

Abstract Types and API Evolution

Abstract types often give us the impression that it will be easier to evolve our API over time. In some ways this is true. If a field returns an interface type, and that we obey to Liskov's substitution principle, adding a new object type which implements an interface should not cause client applications to behave differently. This is true of interfaces, and less true of unions, which allow completely disjoint types. While adding union members and interface implementations is not a breaking change in the strict sense of the word (a client might not immediately break as if we were removing a field definition), it is still a very tricky change to make in most cases, and can almost be considered breaking. It is often referred to as a "dangerous change" for this reason. GraphQL clients are not forced into selecting all union possibilities, or all concrete types on an interface. GraphQL clients should code defensively against new cases, and GraphQL servers should be cautious of adding types that may affect important client logic.

Designing for Static Queries

It is pretty established by now that the query language GraphQL often shines when used directly and explicitly, although it is tempting to use SDKs, query builders, and other tools that look kind of like this:

```
query.products(first: 10).fields(["name", "price"])
```

We lose all visibility on what will be sent to a GraphQL server at runtime. Instead, take the GraphQL query defined directly like this:

```
query {
  products(first: 10) {
    name
    price
  }
}
```

It is very explicit and allows everyone to know right away what data is being asked for, and what the shape of the query will look like at runtime. Not only is the GraphQL language a great choice instead of a query builder, but you should strive to keep your queries **static**. A static query is a query that does not change based on any variable, condition, or state of the program. We know by looking at the source code that it is exactly what the server will receive when the code is deployed. This gives us a lot of advantages over more dynamic queries:

- Just looking at source code gives developers a really good idea of the data requirements of a client.
- We can give operation names to our queries. This greatly simplifies server-side logging and query analysis. For example: `query FetchProducts { products { name } }`

- It opens the door to great tooling on the client-side: think IDE support, code generation, linting, etc.
- It enables the server to save these queries server-side. We'll see this in more detail in Chapter 5.
- Finally, we use the **standard** and **specified** language to interact with GraphQL servers, meaning it is language-agnostic!

Here's a piece of code that builds a query **dynamically**:

```
const productFields = products.map((id, index) => {
  return `product${index}: product(id: "${id}") { name }`;
})

const query = `
  query {
    ${productFields}.join('\n')
  }
`
```

This code has a list of product ids and builds a GraphQL query to fetch the product object associated with each id. At runtime, this query would look something like this:

```
query {
  product0: product(id: "abc") { name }
  product1: product(id: "def") { name }
  product2: product(id: "ghi") { name }
  product3: product(id: "klm") { name }
}
```

The problem here is that:

- The code does not contain the full GraphQL query, so it is hard to see what would actually be sent once the code goes live.
- The query string changes **depending on how many product ids** are in that list at runtime. It uses **field aliases** (`product0`, `product1`, etc.) to do so.

We could use GraphQL concepts instead to avoid the need for constructing a query that always stays the same no matter what, by using variables:

```graphql
query FetchProducts($ids: [ID!]!) {
  products(ids: $ids) {
    name
    price
  }
}
```

This way, the actual query string itself never changes, but the client can fetch as many products as they want simply by providing a different set of variables. For this reason, I recommend offering at least a **plural** version of most fields, and additionally a way to fetch a single entity if needed. A fun fact is that clients can provide a single value to list type arguments in GraphQL:

```graphql
query {
  # This is valid!
  products(ids: "abc") {
    name
    price
  }
}
```

Mutations

Mutations are probably the part that people learning and using GraphQL struggle with the most. They often seem completely different from the query side of things, even if really, mutations are "just" fields like we have on the query side. It makes sense though. Mutations often have to deal with errors, it's often not clear what they should return once they're done with their side effect, and have some strange rules that the query side of things doesn't necessarily have. Let's dive into some design concerns when it comes to mutations.

Input and Payload types

Mutations are simple fields. Just like any other field they can take arguments, and return a certain type. However, rather than return a simple type, a common practice (originally from Relay) is to return what we call "Payload types", a type with the sole purpose of being used as the result of a particular mutation.

```graphql
# An example payload type for a createCheckout mutation
type CreateProductPayload {
  product: Product
}
```

In this example, that payload type simply has a single field, the product that has been created. It is nullable in case the creation fails, which we'll cover in the errors section later on in this chapter. Why don't we simply return a `Product` type for the `createProduct` mutation? The first reason is that this enables us to evolve the mutation over time without having to change the type. Mutation results often require us to return more than just the thing that has been mutated. A payload type allows us to include other information about the result of the mutation, for example, a `successful` field:

```graphql
# An example payload type for a createCheckout mutation
type CreateProductPayload {
  product: Product
  successful: Boolean!
}
```

These payload types should almost always be unique. Trying to share types, as we saw earlier, will lead to problems down the line, especially for mutation payloads. For the input, a similar strategy can be used:

```graphql
type Mutation {
  createProduct(input: CreateProductInput!):
    CreateProductPayload
}

input CreateProductInput {
  name: String!
  price: Money!
}
```

Relay's convention is to use a single, required and unique input type per mutation. The benefits are similar to what we discussed for payload types:

- The input is much more evolvable.
- The mutation can use a single variable on the client-side, which does make it a bit easier to use especially when you start having a lot of arguments.

I don't get crazy about having absolutely one required `input` argument in mutations, but it is definitely a good pattern for consistency, both within your API and with others. However, don't be afraid to use a handful of arguments if needed. We'll see how these input and payloads can be structured in the next few sections.

Fine-Grained or Coarse-Grained

As we saw in the section on Anemic GraphQL earlier on in the chapter, finer-grained, action-based GraphQL mutations offer a ton of advantages. While that sounds great, there's a **darker side**. GraphQL is mostly used as an API interface, which means we're usually interacting with it over the **network**. While the best interface will usually look like what you would design for a programming language method signature, the nature of the network often pushes us to design coarser interfaces for performance reasons. The tradeoff between finer and coarser-grained mutations will often depend on what you think the use case is from your clients (something public APIs make much harder).

Generally, I've found that having a **coarse-grained create mutation** and **finer-grained mutations to update an entity** can be a good rule of thumb. It happens quite often that clients want to create something with a single mutation, but the updates and actions on the created entity might be done in smaller mutations like adding an item, updating the address, etc. Finding the right balance is an art; This is why it is so important to think about the client use case while designing mutations and fields. Overall, this will depend on what you've identified as the real use cases your clients are interested in. We don't want to make the client have to use 5 different mutations or fields for what we could have solved with a single functionality.

While really fine-grained fields and mutations have a ton of advantages, they do push more of the control flow business logic to the client. For example, a common use case might be to create a product, add a label to it, and then modify its price. If this is represented as one action on the UI, the client now has to manage partial failures, retries, etc. to achieve a consistent experience if these actions were all designed as fine-grained actions. Imagine, for example, that the `addLabel` mutation fails. The client needs to recognize that the creation of the product worked, the modification of the price as well, but that the `addLabel` mutation needs to be retried. When this becomes an issue, maybe we've detected an actual use case, and a coarser-grained mutation is starting to make sense for this functionality. In the end, because network calls have a performance cost, we often need to make our operations a bit more coarse-grained than we want, and that's totally ok. It's definitely not a black or white scenario.

Transactions

At some point in your GraphQL journey, you'll hear about a lack of transactions or at least wonder how to do transactions with GraphQL. What do we mean by transactions and GraphQL? Isn't that a database thing? Imagine we have this `addProductToCheckout` mutation from previously. What if a client wanted to add 3 products, but wanted all of the mutations to succeed or none at all?

```graphql
mutation {
  product1: addProductToCheckout(...) { id }
  product2: addProductToCheckout(...) { id }
  product3: addProductToCheckout(...) { id }
}
```

This way of querying has several issues anyways. First, this is the opposite of static queries we covered earlier in the chapter. Clients need to generate query strings with the number of products to add, which is really not great to use for clients. Second, since GraphQL executes all of these mutations one after the other, the second mutation could fail, while the third one could succeed, leading to a really weird state on the client.

A lot of people have been asking for a GraphQL feature to allow running multiple mutations within a single transaction block. Most of the time, this complexity is not needed. Instead, as we discussed in the previous section, designing a coarser grained mutation that contains the whole transaction within one field is almost always a better idea. In this example, it would be as simple as adding an `addProductsToCheckout` mutation that takes multiple items to add. This would solve the static query issue as well as the transaction issue.

If you've got multiple mutations that need to be executed in a sequence, for example adding products to a checkout, updating the billing address, and applying a discount all at once, don't be afraid to include that as an actual use case. Especially if you're dealing with an internal API, feel free to provide specific use cases to solve "transactions" rather than trying to reuse finer grained mutations.

Batch

Another way to do transaction-like operations is to use batching. Batching can mean a lot of things in GraphQL. There's batching as in sending multiple query documents to a server at once, but we can also design a batch mutation within a single query. We've talked about finer-grained mutations vs. coarser-grained mutations, how we want our queries to be static, and how transactions can often be designed as a new mutation field for that use case. One other way to tackle this problem is by building a mutation that takes a few operations as an input:

```
type Mutation {
  updateCartItems(
    input: UpdateCartItemsInput
  ): UpdateCartItemsPayload
}

input UpdateCartItemsInput {
  cartID: ID!
  operations: [UpdateCartItemOperationInput!]!
}

input UpdateCartItemOperationInput {
  operation: UpdateCardItemOperation!
  ids: [ID!]!
}

enum UpdateCartItemOperation {
  ADD
  REMOVE
}
```

In this example, the `updateCartItems` mutation accepts a list of `UpdateCartItemOperation` inputs. Each input describes what kind of operation it represents through the `operation` field, which is an enum of possible operations. The client can then use this mutation to add and remove any number of items in a single mutation:

```
mutation {
  updateCartItems(input: {
    cardID: "abc123",
    operations: [
      { operation: ADD, ids: ["abc", "def"] },
      { operation: REMOVE, ids: ["bar", "foo"] }
    ]
  }) {
    cart {
      items {
        name
      }
    }
  }
}
```

We're lucky that the `ADD` and `REMOVE` both took an `ids` argument as well. In some cases, different operations have different inputs. The best idea to design this would be to use Input Unions. However, at the time of writing this book, they still have not been adopted into the spec. In the meantime, a decent solution is to provide all the fields as optional, and handle this at the resolver/runtime level instead:

```
input UpdateCartItemOperationInput {
  operation: UpdateCartItemOperation!
  addInput: CartItemOperationAddInput
  removeInput: CartItemOperationRemoveInput
  updateInput: CartItemOperationUpdateInput
}
```

It's verbose, and the schema is less expressive since everything is nullable, but it works. This can be useful when batch mutations are really needed until we get input unions in the specification.

Although not standard, you could use directives to indicate that this input type only accepts on of the fields, for example:

```
input UpdateCartItemOperationInput @oneField {
  operation: UpdateCartItemOperation!
  addInput: CartItemOperationAddInput
  removeInput: CartItemOperationRemoveInput
  updateInput: CartItemOperationUpdateInput
}
```

Errors

Errors are a subject that many of us struggle with. Part of this is because there is no one way to do it, and how you handle errors will depend on the context in which your API is meant to be used. We'll try to capture as much of this context as possible while navigating this section. First, let's start with what the GraphQL specification says about errors, and what they look like.

A basic GraphQL error looks like this:

```
{
  "message": "Could not connect to product service.",
  "locations": [ { "line": 6, "column": 7 } ],
  "path": [ "viewer", "products", 1, "name" ]
}
```

We have a `message` which describes the error, `locations` shows where in the query string document this happened, and `path` is an array of string leading to the field in error from the root of the query. In this case, it was the field name on the product with index `1` on the `products` field.

Errors can be extended with more information. To avoid naming conflicts as the specification evolves, the `extensions` key should be used to do so:

```
{
  "message": "Could not connect to product service.",
  "locations": [ { "line": 6, "column": 7 } ],
  "path": [ "viewer", "products", 1, "name" ],
  "extensions": {
    "code": "SERVICE_CONNECT_ERROR"
  }
}
```

Here we added `code` (generally a good idea) to our error for client applications to be able to rely on a stable identifier to handle errors rather than the human-readable `message` string. While query results live under the `data` key of a response, errors are added to GraphQL responses under an `errors` key:

```json
{
  "errors": [
    {
      "message": "Error when computing price.",
      "locations": [ { "line": 6, "column": 7 } ],
      "path": [ "shop", "products", 1, "price" ],
      "extensions": {
        "code": "SERVICE_CONNECT_ERROR"
      }
    }
  ],
  "data": {
    "shop": {
      "name": "Cool Shop",
      "products": [
        {
          "id": "1000",
          "price": 100
        },
        {
          "id": "1001",
          "price": null
        },
        {
          "id": "1002",
          "price": 100
        }
      ]
    }
  }
}
```

In the example above, notice that the `price` field on the product at index `1` returned `null`. Per the spec, fields that have an error should be null and have an associated error added to the `errors` key. This behavior has several implications on how we use these errors in our API. Take mutations, for example:

```
mutation {
  createProduct(name: "Computer", price: 2000) {
    product {
      name
      price
    }
  }
}
```

Let's say we wanted an error for when a duplicate product is added. Any error we add to the mutation field would result in the full mutation response coming back as `null`:

```
{
  "errors": [
    {
      "message": "Name for product already exists",
      "locations": [ { "line": 2, "column": 2 } ],
      "path": [ "createProduct" ],
      "extensions": {
        "code": "PRODUCT_NAME_TAKEN"
      }
    }
  ],
  "data": {
    "createProduct": null
  }
}
```

There are a few downsides to these errors that make them hard to use in these cases:

- The `Payload` type for mutations we saw previously was great to encode metadata about the mutation. In this case, because the field must return `null`, we lose the possibility of sending back data to the client for this mutation, even if there was an error.
- The information on errors is limited, which means servers will usually need to add additional keys within an `extension` key. If we add it to the error itself, we risk the specification clashing with the fields we added.
- The `errors` payload is outside the GraphQL schema, meaning clients don't get any of the benefits of the GraphQL type system. That means our errors are harder to consume but also harder to evolve.

- The `null` propagation we saw in the section on nullability needs to be always on top of our mind since errors could end up having catastrophic repercussions on a query if most fields are non-null.

This all makes sense when we understand that GraphQL errors were originally designed to represent exceptional events and client-related issues, not necessarily expected product or business errors that need to be relayed to the end-user. It's helpful to divide errors into two very broad categories to understand what goes where:

- Developer/Client Errors: Something went wrong during the query (Timeout, Rate Limited, Wrong ID Format, etc.). These are often errors that the developer of the client application needs to deal with.
- User Errors: The user/client did something wrong (Trying to pay for a checkout twice, Email is already taken, etc.) these things are part of the functionality our API provides.

The GraphQL "errors" key we just covered is a great place to capture developer/client errors. It is meant to be read by developers and handled by the GraphQL client. For user-facing errors that are part of our business/domain rules, the current best practice is to look at designing these errors **as part of our schema** rather than treating them as exceptions/query level errors. Let's take a look at some ways of achieving this.

Errors as data

The easiest way to design errors as data is to simply add fields that describe possible errors in our payload types:

```
type SignUpPayload {
  emailWasTaken: Boolean!
  # nil if the Account could not be created
  account: Account
}
```

Clients can then consume this information as they wish. Now that way of handling errors can work with purely internal use cases but makes it hard to handle errors generically and to be consistent across mutations. A possibly better approach is for payload types to include something like a `userErrors` field:

```
type SignUpPayload {
  userErrors: [UserError!]!
  account: Account
}

type UserError {
  # The error message
  message: String!

  # Indicates which field cause the error, if any
  #
  # Field is an array that acts as a path to the error
  #
  # Example:
  #
  # ["accounts", "1", "email"]
  #
  field: [String!]

  # An optional error code for clients to match on.
  code: UserErrorCode
}
```

Something that makes errors as data a bit annoying, as compared to error mechanisms like status codes, is that clients don't **have to query the `userErrors`** field. This means that some clients might be getting a null account back, but have no idea why this happens. Part of the solution is educating clients on your APIs best practices, but there is no guarantee that clients will be aware of errors if they don't specifically include that field. Note that clients could also ignore the errors in the "errors" GraphQL response, but at least they are described by the GraphQL specification, meaning a lot of clients will already have error detection mechanisms in place for those types of errors.

Union / Result Types

Another approach that is getting some popularity is also an "errors as data" approach, but instead of using a specific field for errors, this approach uses union types to represent possible problematic states to the client. Let's take the same "sign up" example, but design it using a result union:

```graphql
type Mutation {
  signUp(email: string!, password: String!): SignUpPayload
}

union SignUpPayload =
  SignUpSuccess |
  UserNameTaken |
  PasswordTooWeak
```

```graphql
mutation {
  signUp(
    email: "marc@productionreadygraphql.com",
    password: "P@ssword"
  ) {
    ... on SignUpSuccess {
      account {
        id
      }
    }

    ... on UserNameTaken {
      message
      suggestedUsername
    }

    ... on PasswordTooWeak {
      message
      passwordRules
    }
  }
}
```

As you can see, this approach has several advantages. The union type here very well describes what could happen during the execution of the mutation, and each case is strongly typed using the GraphQL type system. This allows us to add custom fields to each error scenario, like a `suggestedUserName` when a user name was taken or a list of password rules when the password provided was too weak.

It exemplifies what both the approaches we saw are aiming to convey: some API errors are meant to be exposed as use cases just like any other field or type, and clients should be able to consume them the same way.

So, which error style should you pick?

Honestly, as long as "user errors" are well defined in the schema, I don't have a strong opinion on how they should be implemented. Even simple fields like `emailIsTaken` on the payload types can be used if you're working with a low number of clients / getting started. Using unions is definitely a very expressive way to define errors. One thing to keep in mind is that clients must be coding defensively against new types of errors in their clients. For example:

```
mutation {
  createProduct {
    ... on Success {
      product {
        name
      }
    }
    ... on ProductNameTaken {
      message
    }
    # What if ProductPriceTooHigh gets added?
  }
}
```

We must make sure clients can handle a new possible type like `ProductPriceTooHigh`. In a world where our schema is versioned and where client applications are compiled against our schema and make sure our case checking is exhaustive, this would be perfect! However, most of us do not have this chance, especially not in the world of web APIs. This is why I think the union type technique is great, and in theory perfect (in a compiled language with exhaustive case checks), but can possibly fall short when used in GraphQL.

At this point, a possible advantage to a `userErrors` list in these cases is that the client will get all new errors within that same field they already selected. This means that even though they might not handle the error perfectly, they can display it if they want and log the error message for easier debugging. It might encourage a bit more of a generic approach to handling errors on the client.

One approach we can use to help clients handle new error scenarios is by using an interface type to define an error contract:

```
interface UserError {
  message: String!
  code: ErrorCode!
  path: [String!]!
}

type DuplicateProductError implements UserError {
  message: String!
  code: ErrorCode!
  path: [String!]!
  duplicateProduct: Product!
}
```

This way, clients can always select `message`, `code` and `path` on errors, but can select error specific fields as well:

```
mutation {
  createProduct(name: "Book", price: 1000) {
    product {
      name
      price
    }
    userErrors {
      message
      code
      path
      ... on DuplicateProductError {
        duplicateProduct
      }
    }
  }
}
```

The same technique can be applied to unions and it helps users by allowing them to match on the interface type rather than having to cover all the possible concrete types:

```
mutation {
  createProduct(name: "Book", price: 1000) {
    ... on CreateProductSuccess {
      product {
        name
        price
      }
    }
    ... on DuplicateProductError {
      duplicateProduct
    }
    ... on UserError {
      message
      code
      path
    }
}
```

All in all, both the union approach and the `userErrors` approach can be effective. What is common between both of these approaches is that we treat errors as data in our schema. However, the `userErrors` approach is often easier to programmatically populate at runtime. Sometimes in existing complex applications, validation rules are hard to explicitly define and require more generic handling of errors:

```
user_errors = errors_from_model(product)
```

On the other end, you can define types and clearly define the scenarios using the schema which is a great advantage. Just keep in mind that users need to make queries in a way that handles new cases if schema evolution is important to you. As we saw earlier in the chapter, the union approach also allows us to avoid designing impossible states. With the `userErrors` approach, it is sometimes unclear what fields will be null when there are errors.

In all these cases, nothing is forcing clients to either match against error types in the union case, or to select a `userErrors` field on the payload type. This is a problem that is yet to be solved in the GraphQL community and that currently requires documentation to help users understand which fields/types they need to look out for when executing mutations and fields that could return errors. With GraphQL errors, we at least know the errors are likely to be correctly handled by clients.

Schema Organization

A GraphQL schema is one big graph of possibilities from which clients have to select their requirements. For this reason, some people are tempted to ask about organizing a schema in a way that makes finding use cases easier, or at least groups similar concepts together.

Namespaces

Many people have been asking for a namespacing mechanism for GraphQL schemas. So far no proposal has advanced very far into the specification. If you follow the advice we covered in naming earlier in this chapter, I rarely see this becoming a huge issue. If you are specific enough in naming, situations where namespacing is absolutely needed will be rare.

However, if you must namespace things in a very explicit way, I recommend using a naming strategy, like prefixes for example:

```graphql
type Instagram_User {
  # ...
}

type Facebook_User {
  # ...
}
```

From what I've heard and seen, most of the namespacing requests come from developers using strategies like **schema stitching**, which we'll cover in Chapter 8. Schema stitching allows merging different schemas which opens the door to naming conflicts. Again, this can often be solved by proper naming and can be assisted by build-time tools instead of a specific GraphQL feature. Remember that the resulting GraphQL schema and the way you build that schema server-side are two completely different issues. We can still use namespaces, modules, and reusable functions on the server-side to help with code organization.

Mutations

Some teams struggle naming mutations. Should you name your mutation `createProduct` or `productCreate`? For example, back when I was at Shopify, the team decided to go the `productCreate` route to allow for better discoverability, since related mutations would be grouped in the SDL / introspection / GraphQL. That makes sense, but for me, it always felt sad that we were going for a less readable name just for the sake of organization.

Don't be afraid to use specific and readable names for your mutations like `addProductsToCart` rather than things like `cartProductAdd`. Tools can help with discoverability anyways. One idea I've been playing with is to use a `tags` directive to help with that kind of grouping for documentation and other tools:

```
type Mutation {
  createProduct(...):
    CreateProductPayload @tags(names: ["product"])

  createShop(...):
    CreateShopPayload @tags(names: ["shop"])

  addImageToProduct(...):
    AddImageToProductPayload @tags(names: ["product"])
}
```

Another idea is to use parent fields or "namespaces" to group similar fields under parent fields like `shop { create(...) { id } }` but the specification and tooling are not 100% clear on that kind of usage and I would probably not recommend using that approach at the moment. Top-level fields on the mutation root are the only fields expected to have a side effect. This means that in mutations grouped like these:

```
mutation {
  products {
    deleteProduct(id: "abc") {
      product
    }
  }
}
```

The `deleteProduct` should actually be considered a read-only field in theory. Personally, I don't recommend going this route even though it might technically work in the server implementation you are using.

Asynchronous Behavior

Sometimes our APIs are not synchronous. They can't return a result right away as something needs to process in the background. In REST and HTTP based APIs, the most common practice to handle this is the `202 Accepted` status code. However, using this with GraphQL is trickier because it might be that only a part of our request is asynchronous while the rest can return right away. There are a few approaches to model this. The first thing we can look at is modeling the asynchronous concept explicitly, for example, if we're dealing with payment processing, we could design a pending payment as a type, within a union type:

```
type Query {
  payment(id: ID!): Payment
}

union Payment = PendingPayment | CompletedPayment
```

We can also opt for a single type instead of a union:

```
type Operation {
  status: OperationStatus
  result: OperationResult
}

enum OperationStatus {
  PENDING
  CANCELED
  FAILED
  COMPLETED
}
```

Another solution is to handle these cases more generically, with a `Job` type and concept. This is what Shopify went for in their admin GraphQL API. The idea is pretty clever. A job is identifiable through a global ID, and simply contains two other fields:

- A `done` boolean type, which indicates if the asynchronous job has been completed or is still pending.
- The fun part: A `query` field which returns the `Query` root type. This helps clients query the new state of things after the job has been completed.

Data-Driven Schema vs Use-Case-Driven Schema

So far we've talked a lot about designing well-crafted GraphQL schemas that express real use cases. When in doubt, designing a GraphQL schema for behaviors instead of for data has been my go-to rule. While this usually leads to a great experience when consuming an API, there are other distinct uses for an API. The GitHub GraphQL API is a good example of a schema designed with business/domain use cases in mind. The schema is good for an application looking to interact with the GitHub domain and consume common use cases: things like "opening an issue", "commenting", "merging a pull request", "listing branches", "paginating pull requests", etc.

However, certain clients, when seeing GraphQL's syntax and features, see great potential for getting exactly the data they require out of a business. After all, the "one GraphQL query to fetch exactly what you need" thing we hear so often may be interpreted that way. For example, take a comment analysis application that needs to sync all comments for all issues every 5 minutes. GraphQL sounds like a great fit to do this: craft the query for the data you need, send one query, and profit. However, we hit certain problems pretty quickly:

- Most APIs paginate their resources as they are not made for mass consumption of data, but for displaying results to a human, for example. When what you're looking for is pure data, paginated fields make that much more difficult.
- Timeouts: Most GraphQL API providers don't want gigantic GraphQL queries running for too long, and will aggressively use timeouts and rate limits to block that from happening (We'll learn more about this in Chapter 4). However, purely data-driven clients might need to make pretty large queries to achieve their goals. Even though it's a valid use case and not an excessive use scenario, there is quite a high chance that queries could be blocked if they query thousands of records.

Pretty hard to deal with, right? On one end, clients that purely have a data-driven use case may have a legitimate reason to do so, but on the other end, your GraphQL API might not be designed (and nor should it be) for this purpose. In fact, this is not necessarily a GraphQL problem. Most APIs out there today are mostly aimed at building use-case driven clients, and would be hard to deal with when wanting to sync a large amount of data (large GraphQL requests, batch HTTP requests, or tons of HTTP requests).

When the use case is actually about data, we have to consider other ways.

Asynchronous GraphQL Jobs

Most large requests of that kind take time to compute and executing them during a request is simply not feasible for API providers. Another idea is to schedule a query for an API provider to execute asynchronously, and get the results at a later time. Practically, we could implement it by having an endpoint to register async queries:

```
POST /async_graphql
{
  allTheThings {
    andEvenMore {
      things
    }
  }
}

202 ACCEPTED
Location: /async_graphql/HS3HlKN76EI5es7qSTHNmA
```

And then indicating to clients they need to poll for the result somewhere else:

```
GET /async_graphql/HS3HlKN76EI5es7qSTHNmA

202 ACCEPTED
Location: /async_graphql/HS3HlKN76EI5es7qSTHNmA
```

```
GET /async_graphql/HS3HlKN76EI5es7qSTHNmA

200 OK
{ "data": { ... } }
```

As another example, once again, Shopify has a great example of Asynchronous GraphQL job in practice, which they call "Bulk Operations".. Although they use the same schema for these operations as with their synchronous GraphQL API, they can auto paginate resources, and you don't risk getting rate limited as much or facing query timeouts. This is a great way to support longer-running, more data-oriented use cases.

Summary

Great API design goes beyond GraphQL. Most of the best practices we've seen in this chapter apply to things like HTTP APIs, library design and even UI design in some cases. We've seen many principles in this chapter, but I think there are four main points you should remember when building your GraphQL schema.

- First, use a design-first approach to schema development. Discuss design with teammates that know the domain best and ignore implementation details.
- Second, design in terms of client use cases. Don't think in terms of data, types, or fields.
- Third, make your schema as expressive as possible. The schema should guide clients towards good usage. Documentation should be the icing on the cake.
- Finally, avoid the temptation of a very generic and clever schema. Build specific fields and types that clearly answer client use cases.

If you follow these general practices, you'll already be well on your way to a schema that clients will love to use, and evolve well over time.

Implementing GraphQL Servers

In the last few chapters, we saw how GraphQL came to be and how the query language and type system allows multiple clients to consume different use cases through possibilities exposed by the server. We also covered what we should be thinking about when designing a GraphQL schema. While we discussed the concepts, we didn't talk about how GraphQL servers are built and implemented in practice. In this chapter, we'll see the main concepts in implementing GraphQL servers and a few examples. Every single languag- specific library has a different way of doing things, so we'll be focused more on principles rather than specific ways of doing things.

GraphQL Server Basics

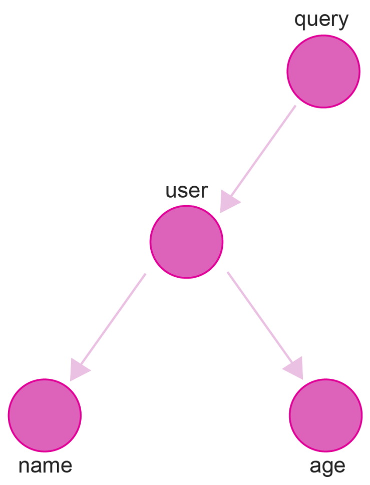
Building a GraphQL server is simple in theory, but quite tricky in practice. If you're familiar with the basic concepts of a GraphQL server, feel free to skip to the next section. At its core, building a GraphQL server mainly requires three main things:

- A type system definition (What we were designing in Chapter 2)
- A runtime execution engine to fulfill the requested queries according to the type system. (What this chapter is all about)
- In most cases: an HTTP server ready to accept query strings and variables.

Almost every single language has a GraphQL server implementation, and all of them let us achieve these two elements in different ways. A user will define both the type system and runtime behavior for an API, and the library will usually take care of implementing the GraphQL spec including the execution algorithm. To fulfill queries, we need more than a type system, we need behaviors and the data behind this type system. In GraphQL the concept used to fulfill data for a certain field is called a **resolver**. At their core resolvers are really just simple functions:

```
function resolveName(parent, arguments, context) {
  return parent.name;
}
```

A resolver is in charge of resolving the data for a single field. The GraphQL engine will call the resolver for a particular field once it gets to this point. In fact, the basic execution of a GraphQL query often resembles a simple depth-first search of a tree-like data structure:



At every step or node of a GraphQL query, a GraphQL server will typically execute the associated resolver function for that single field. As you can see in the example above, a resolve function usually takes 3 to 4 arguments. The first argument is usually the parent object. This is the object that was returned by the **parent resolver function**. If you take a look a the image above, this means that the `name` resolver would receive the user object the `user` resolver returned. The second argument are the arguments for the fields. For example, if the field user was called with an `id` argument, the resolver function for the user field would receive the value provided for that id argument. Finally, the third argument is often a `context` argument. This is often an object containing global and contextual data for the particular query. This is often used to include data that could potentially be accessed by any resolver.

To put this into context, let's say we had this query:

```
query {
  user(id: "abc") {
    name
  }
}
```

You can imagine the GraphQL server would first call the user resolver, with our "root" object (which varies based on implementations), the `id` argument, and the global context object. It would then take the result of the `user` resolver and use it to call the `name` resolver, which would receive a user object as a first argument, no extra arguments, and the global context object.

With a type system in place, and resolvers ready to execute for every field, we've got everything needed for a GraphQL implementation to execute queries. **If you have never implemented any GraphQL server until now, I'd encourage you to play around with your favorite language implementation of GraphQL and come back to this when you're ready**. For now, we're moving on to slightly more advanced concepts and considerations.

Code First vs Schema First

Probably the biggest debate in the last few years, when it comes to implementing GraphQL servers, is whether a "code-first" or "schema-first" approach is best. These approaches refer to the way the type system is built on the server-side.

Schema First

With schema-first, developers mainly build their schemas using the Schema Definition Language (SDL) that we covered in the previous chapter. In JavaScript, using the reference implementation, a schema-first approach looks like this:

```
var { graphql, buildSchema } = require('graphql');

var schema = buildSchema(`
  type Query {
    hello: String
  }
`);
```

A schema string is passed to `buildSchema`, which builds a GraphQL schema in memory. While having a schema is great, if it doesn't have any logic on what this `hello` field should return when a client requests it, it is pretty useless. Schema-first approaches usually let you define **resolvers**, by mapping them to fields and types defined using the SDL. Here's what it looks like, again with the GraphQL JavaScript reference implementation:

```
// Define a resolver object that can augment
// the schema at runtime
var root = {
  hello: () => {
    return 'Hello world!';
  },
};

// Run the GraphQL query '{ hello }' and print out the response
// Use the `root` resolver object we created earlier.
graphql(schema, '{ hello }', root).then((response) => {
  console.log(response);
});
```

The schema-first approach is not to be confused with a design-first approach, something that is almost always a good idea when it comes to building APIs, as we covered in the previous chapter on schema design. By schema-first, we mean that we are **building the schema** using the SDL directly, rather than defining it purely with structures available in the programming language we are using. This will get clearer as we explore code-first.

There are great benefits to a schema-first approach. One of them is the fact that we're building the schema using a common language, the SDL itself. This lets engineers know with great confidence what the resulting GraphQL schema will be. Not only that, but it forces the schema developers to think in GraphQL, rather than think straight away about implementation.

The SDL is a great tool in general. It serves as a common language between teams using different languages, or simply between clients and servers. But it also serves as a common language between computers: a ton of tools operate on the SDL itself, making them language agnostic. This is a big advantage, especially if GraphQL tools don't operate well with the programming language you're building with.

The SDL is a great tool, but it can't do everything. For example, as we just saw in the previous example, the SDL itself provides no mechanism to describe the logic that should be executed when a field is requested. It makes sense, the GraphQL schema language is made to describe an interface, but it's not a powerful programming language that we need to execute network calls, database requests, or any logic we usually need in an API server.

The fact that we need to separate the schema description and what happens at runtime can be a challenge with a schema-first approach. When a schema grows large enough, it can be a big challenge to ensure the type definitions and their mapped resolvers are indeed valid. Any change in the schema must be reflected in the resolvers, and that is true for the opposite as well.

Finally, the other issue is that using the SDL makes it harder to define reusable tooling and type definition helpers. As schemas grow large and more and more team members contribute to the schema, it's often useful to encapsulate schema definition behavior in reusable functions or classes rather than typing it out entirely, which opens the schema to inconsistencies.

These problems are all mostly solvable with careful tooling, but they usually require **code** to be involved at some point to solve them. So what if we just used code to start with?

Code First

A code-first approach is an alternative to that schema-first approach. Instead of using the SDL, we use the programming language primitives we have at our disposal. Here's an example from the GraphQL Ruby library that might make this clearer:

```ruby
class PostType < Types::BaseObject
  name "Post"

  description "A blog post"

  field :title, String, null: false

  field :comments, [Types::CommentType], null: true,
    description: "This post's comments"

  def title(post, _, _)
    post.title
  end

  def comments(post, _, _)
    object.comments
  end
end
```

As you can see, GraphQL Ruby represents GraphQL object types using **Ruby classes**. Represented as the SDL, this class would look like this:

```graphql
type Post {
  title: String!
  comments: [Comment]
}
```

In this case, using Ruby to define the schema provides engineers working on the

GraphQL schema with all the tools they are used to with Ruby, for example, including a module for common functionality or programmatically define types. A common example we used at GitHub is generating **connection types**. Connection types are a way to do pagination in GraphQL. They usually require a lot of boilerplate to get started:

- Defining a `XConnection` type, like `PostConnection`
- Defining an edge type, like `PostEdge`

Using a code first approach allows providers to serve different versions of an API without building an entire schema. For example, instead of having to type the full connection types for all our paginated list, in a code-first scenario, we can simply write code to automate this type generation:

```ruby
# Connection.build creates a connection type as
# well as the edge type for paginating comments
field :comments, Connection.build(Types::Comment)
  description: "This post's comments"
```

The sky is the limit for what we can build with code-first since we're simply playing with code rather than the SDL. The other big advantage is that usually, code-first approaches will have you define both the schema and the resolvers, the interface and the runtime logic, in the same place. This is a big thing when it comes to maintenance, especially when you're dealing with hundreds to thousands of types and fields.

However, code-first means that tools that operate on the SDL can't understand your schema definition anymore. It also means that we can abstract GraphQL so much that it's not clear what becomes exposed to users anymore. This needs to be taken into consideration, and we can solve that as well.

SDL Design, Code Implementation, SDL Artifact

My favorite approach is a hybrid one. The SDL is a great tool to discuss schema design before going forward with the implementation. I prefer using a GitHub issue, or whatever issue tracking software you use to discuss a design first. The SDL allows anyone to see the proposed schema and is usually much faster to write than doing the same in a programming language.

Once satisfied with the design, we can move towards implementation using a code-first approach. This lets us use powerful tools, but also co-locate definitions and resolvers. Now, we talked about how the programming language abstractions might make it harder to see what the resulting schema could be. One of the most powerful patterns I saw both at Shopify and GitHub is having an **SDL artifact generated from code definitions**. This artifact should be committed to your version control system and should be tested against your code to always be in sync.

This gives us kind of a best-of-both-worlds solution. We use the power of our programming language to define schemas more efficiently, but we keep the "source of truth" the SDL gives us. Having the SDL being generated from code as an artifact allows reviewers and teammates to quickly understand what is going on with GraphQL changes. It also lets us use any tools that operate on SDLs, even though we defined it in another language in the first place. This is highly recommended if you go with a code-first schema definition approach.

Annotation-Based Implementations

Somewhere in the battle between schema-first and code-first approaches that we've seen so far hides another good approach to building schemas: an annotation-based approach. It is allegedly how Facebook's GraphQL Schema is built and while it's closer to a code-first approach, it's not exactly the same.

An annotation-based schema implementation usually uses your domain entities, objects, classes, etc. and annotates them to generate a GraphQL schema out of them. The big advantage here is that the interface layer is tiny. You don't need hundreds of objects defining a GraphQL schema. Instead, you may simply reuse the domain entities you already have.

The danger with this approach is that it depends on your "entities" making a good API in the first place. If all you have are ActiveRecord objects closely coupled to your database schema or implementation details, an annotation-based approach risks making your resulting API not so great or stable. However, if you do have well-defined entities that represent your domain without being coupled to implementation details like the database, this might be a really great approach.

A good OpenSource example of this is graphql-spqr, a Java library for building GraphQL servers:

```java
public class User {

    private String name;
    private Date registrationDate;

    @GraphQLQuery(
      name = "name",
      description = "A person's name"
    )
    public String getName() {
        return name;
    }

    @GraphQLQuery(
      name = "registrationDate",
      description = "Date of registration"
    )
    public Date getRegistrationDate() {
        return registrationDate;
    }
}
```

Java is no stranger to annotation based approaches in general, which makes it work quite nicely. Overall, it's incredibly similar to a code-first approach, but the annotation-based approach may help if you already have well defined domain objects.

Generating SDL Artifacts

Generating SDL artifacts as we talked about in a code-first context is a great practice, but not necessarily always straightforward. Fortunately, some GraphQL libraries offer this functionality out of the box. For example, GraphQL-Ruby has a Schema::Printer class that can receive a schema Ruby object, and turn it into an SDL string:

```ruby
# Print the SDL document based on MySchema
# An instance of GraphQL::Schema
GraphQL::Schema::Printer.print_schema(MySchema)
```

In JavaScript land, GraphQL-JS contains a utility method called printSchema, which does pretty much the same, except it is much less customizable than the Ruby version:

```
import { printSchema } from 'graphql';

// Print the SDL document based on schema
// A variable of type GraphQLSchema
printSchema(schema)
```

Any language that supports schema printers should give you a good base. Any new field or type added through a code-first approach gets reflected in the artifact. Your documentation generator, linter, and any other tool can operate on that source of truth instead of analyzing code. The other option is an introspection query, but that's done usually done at runtime, and doesn't necessarily reflect the whole schema, but just whatever token or user's view of the schema. A better approach is to print the whole schema. Instead of hiding feature flagged fields, or hiding partner only types, we print everything and annotate them with metadata, so it's clear when viewing the SDL that these types are not available to everyone:

```
directive featureFlagged(name: String!)
  on FieldDefinition | TypeDefinition

type Query {
  viewer: User
}

type User {
  name: String
  secretField: SecretType @featureFlagged(name: "secret")
}

type SecretType @featureFlagged(name: "secret") {
  name: String!
}
```

This approach is great to improve confidence in how changes affect the schema as a whole. Not only that but this can be the basis of all developer tooling including linting, breaking change detectors, documentation generators, and so much more. I highly recommend checking in this schema artifact into source control, meaning developers making changes quickly see the resulting schema structure and a single source of truth of the schema is versioned for you.

The bad news is that you'll probably need some custom printing logic to pull this off unless your metadata solution also adds directive definitions under the hood, which certain implementations can do. GraphQL Ruby makes custom printers really easy, but GraphQL-JS has by default only a simple `printSchema` function.

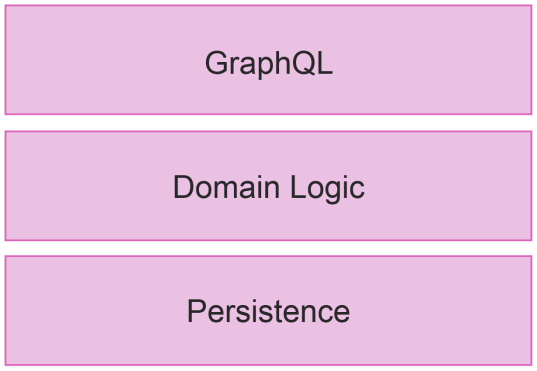
Fortunately, that file is not that large, which means you could potentially implement your own.

Summary

I highly encourage you to build your schema using a code-first approach when you can, but sometimes the specific programming language you choose will dictate the approach depending on what the most mature library picked as a schema definition scheme. For example, GraphQL Ruby, Sangria (Scala), and Graphene (Python) already favor code-first, but in JavaScript land, it depends on your choice of library. I recommend using tools like GraphQL Nexus or using GraphQL-JS directly. Annotation-based approaches like SQPR and type-graphql can also be great choices, but be careful you are not tying your GraphQL schema to implementation details.

Resolver Design

No matter how you're building a schema, you will have to write resolvers at some point. The resolver pattern of GraphQL's execution engine is great. It helps teams add new use cases by working on their isolated resolver and makes it simple to think about the logic of a single field. However, it can be overused. At its core, GraphQL is an **API interface**. It's an interface to your domain and business logic, and in most cases, should not be the source of truth for your business.



So how does that actually look practically? A great resolver often contains very little code. It deals with user input, calls down to our domain layer, and transforms the result into an API result. Now describing what that "domain" layer should look like is a whole different story and would take an entire book to describe. However, in general, encapsulating behaviors, at least outside of our persistence and interface layers will be a great start. It's really tempting to start handling a ton of different validation rules in there, but try to make resolvers as "dumb" as possible. Chances are your GraphQL API is not the only entry point to your business. If you have a UI, a REST API, or even internal RPC calls across micro-services, you want all these things to be consistent. If you start handling too much logic at the GraphQL layer, it risks becoming out of date or straight out wrong over time.

Beware of the Context Object

The `context` argument that most GraphQL implementations allow you to use in resolvers is very useful. For example, it is incredibly useful to store "request" level information like values of certain headers, the request IP, authentication and authorization data, etc. These are totally valid uses of `context`, but using it to alter runtime behavior can be very detrimental to building a stable API. Every single conditional we add around context makes the API less predictable, less testable and harder to cache. Let's take a look at the following resolvers:

```ruby
def user(object, arguments, context)
  if BLACKLIST.includes?(context[:ip])
    nil
  else
    getUser(arguments[:id])
  end
end
```

While this code is not necessarily wrong, having too many of these conditionals relies on the same exact context to be provided on every call. For example, a query executed in a totally different context, maybe an internal call or an inter-service call where the `ip` is not provided, would result in broken resolver logics.

Another pattern to avoid, if possible, is **mutating** the context object during the query execution like this:

```ruby
def user(object, arguments, context)
  user = getUser(arguments[:id])
  context[:user] = user
  user
end
```

This could make your resolvers order dependent and almost surely lead to issues as your schema evolves or as users execute query patterns you haven't thought of. It's very hard to avoid using `context` entirely but try to be very selective of what goes in there. A good idea might also be to make certain parts of `context` to be required before executing a query.

Lookaheads and Order Dependent Fields

Some libraries allow you to fetch information on what fields are being queried under the field that is currently being resolved by your function. Be very careful with this information. It may be tempting to preemptively load data for an order's products for example. There are problems with writing resolver logic that depends on the query shape or the child fields being queried under it:

- GraphQL fields on the query root can be executed in parallel; you probably don't want to lock yourself forever by having resolver logic that is not parallelizable.
- As your schema evolves, new queries that your resolvers did not expect can start making an appearance, meaning the logic you wrote for handling subqueries is now out of date and most likely wrong.

Trying to be too smart about the execution of a query will often make things worst. The beauty of the resolver pattern is that it does one thing, and does it really well. Try to keep its logic isolated. In general, don't assume where a type or field might be used in a query. Chances are it will be reused in a different context, and the main thing that you should rely on is the object it receives. Except for mutation fields, resolvers work best as **pure functions**; they should behave as consistently as possible and have no side effects when queried on the query root. This means avoiding mutating the query context at all costs and not depending on the execution order at all.

Summary

Writing great resolvers is easier said than done, but we can focus on some core principles to make sure we're heading towards the right direction:

- Keep as much business logic out of the GraphQL layer.
- Keep the `context` argument as immutable as possible.
- Don't depend on your fields being called in a certain order, or assume that a certain value in `context` was filled in by another field.

Schema Metadata

Any sufficiently complex GraphQL API will at some point need to add metadata or any extra information to its type system. While the type system allows us to describe fields, their types, and their nullability, it can't do everything. There's a ton of other things we might want to attach to schema members. For example, GitHub's GraphQL framework allows engineers to tag types with the schema version (internal or public), whether the type is under development, if the type is hidden behind a special feature flag, which oauth scopes are needed to access a certain schema member, and a lot more. We'll cover some of these in more detail later.

Metadata is easy when you're using a code-first approach, we can do it any way we want. GraphQL Ruby allows us to define base classes from which we can extend. This lets us define our own DSL and metadata, for example:

```ruby
class PostType < MyBetterObjectType
  name "Post"
  description "A blog post"

  under_development since: "2012-07-12"

  schema :internal

  scopes :read_posts

  # Field definitions
end
```

GraphQL Nexus allows this sort of metadata on most schema members as well, for example setting a `complexity` attribute on a field `id`:

```js
export const User = objectType({
  name: "User",
  definition(t) {
    t.id("id", {
      complexity: 2,
    });
  },
});
```

Metadata with a schema-first or SDL-first approach is commonly done through **schema directives**. These directives are then interpreted by the framework to allow for custom use cases. So instead of having a custom Ruby DSL like the example we just saw, the same metadata could be defined like this in a SDL-first approach:

```
type Post
  @underDevelopment(since: "2012-07-12")
  @schema(schema: "internal")
  @oauth(scopes: ["read_posts"]) {
  title: String!
  comments: [Comment]
}
```

The story around how to publish this metadata for other machines to consume or let clients fetch it through introspection is complicated. At the time of writing this book, there are still discussions around how to do this best. This is because schema directives are not currently exposed through introspection requests. My recommendation around this problem would be to encode this metadata in the SDL you generate, even though the actual introspection queries don't support it. This at least lets tools consume this metadata out-of-band. This difference between the SDL dump and the introspection payload is a bit annoying at the moment and hopefully will be addressed in the spec eventually.

Schema metadata is a great way to guide developers adding new functionality to the API to reuse great patterns. This lets us write reusable code that executes based on some metadata rather than asking for all developers to copy code around. For example, an `oauth_scope` metadata attribute not only lets us publish this to the SDL and generate docs from, but also automate the checks for all resolvers defining this metadata. This is a must for anyone building a GraphQL platform that might be extended by multiple developers.

Multiple Schemas

Some of us have different schemas to maintain. For example, Shopify has both a **storefront** and an **admin** API. GitHub has both an **internal** and **public** API. Maintaining multiple schemas can be tricky. Should these schemas be completely separated and different or should they share common concerns? There are two main approaches when it comes to maintaining and publishing multiple GraphQL schemas. The first approach is to build different schemas at **build time** and the second approach is to build different schemas at **run time**.

The build time approach is the simplest. We simply build different schemas for different use cases. This works best when schemas are fairly different, but we can still share object type definitions and really any code common to your GraphQL platform. For example, this approach can work quite well:

```
+-- admin
|   +-- order.rb
|   +-- schema.rb
+-- storefront
|   +-- collection.rb
|   +-- product.rb
|   +-- schema.rb
+-- common
|   +-- product_image.rb
```

The big advantage of this approach is that it's very easy to see what schemas are being built. It also ensures our schemas can easily evolve in any way they want. For this reason, make sure you don't reuse type definitions unless you're certain the types will stay the same. Again, you may share code, but avoid sharing actual GraphQL types.

The runtime approach comes in useful when the most part of the schemas is the same, but they vary in small ways. For example, a public API may be a subset of an internal API, or you may offer only certain types to certain users. In that case, schema metadata comes in very useful to avoid building two very similar schemas and having to maintain types in duplicate. Here's an example using GraphQL Nexus, which allows us to define extra config on object types:

```
const User = objectType({
  name: "User",
  // Annotate this type as INTERNAL only.
  schema: INTERNAL,
  definition(t) {
    t.int("id", { description: "Id of the user" });
  },
});
```

Now the missing part is that we need the capability to hide or show fields based on a certain request. This is most commonly referred to as **schema visibility filters**, or **schema masks**.

Schema Visibility

When developing a new REST endpoint for select partners, or simply when building new functionality behind a feature flag, we generally don't want this endpoint to be publicly accessible. Usually, anyone who is not supposed to be aware of that new resource would get a `404 Not Found`. We wouldn't want to return a 403 Unauthorized, because that would leak the **existence** of that resource. While we want to block usage unless feature flagged, we also don't want to risk leaking an upcoming resource.

It is very common for large GraphQL APIs to need the same mechanism, except we deal with a single endpoint. Instead of hiding the existence of a resource, we want to hide subsets of the schema like fields, types, or really any schema member. This is commonly referred to as schema visibility or schema masking. Schema visibility is not only useful for publishing new features behind feature flags or sharing schemas only with certain partners, but we'll see in the next section that it can help us maintain multiple versions of a schema by simply applying certain visibility masks at runtime.

Note that this is very different from your typical authorization. Authorization can usually be implemented within a field resolver, returning `null` and/or an error. With visibility, we don't event want clients to see that fields or types **exist** in the schema. GitHub's schema previews is a good example of visibility in action. New schema features that are only accessible if a special preview header is passed along with the query. If a client doesn't have the header, it will receive a "Field does not exist error".

At the time of writing this book, schema visibility mechanisms are unfortunately not implemented in all libraries. GraphQL-JS is lacking this feature, but both GraphQL Ruby and GraphQL Java give developers the ability to restrict visibility on certain schema members. If your library of choice has no support for this, and you're convinced this is an important feature after reading this chapter, I encourage you to either implement it or open an issue for it!

Combining schema metadata with visibility is where things start being really powerful. For example, In GitHub's implementation, developers can annotate types with a feature flag:

```ruby
class SecretFeature < ProductionReadyGraphQL::BaseType
  name "Secret"
  description "Do no leak this."

  feature_flagged :secret_flag

  # Field definitions
end
```

Then, at runtime, a visibility filter is applied to the execution of the query, only allowing users that are on-boarded to this feature flag to see that this type exists. GraphQL Ruby is smart enough to hide all fields that are of the feature flagged type, and hide those too.

```ruby
class FeatureFlagMask
  # If this returns true, the schema member will be allowed
  def call(schema_member, ctx)
    current_user = ctx[:current_user]

    if schema_member.feature_flagged
      FeatureFlags
        .get(schema_member.feature_flagged)
        .enabled?(current_user)
    else
      true
    end
  end
end

MySchema.execute(query_string, only: FeatureFlagMask)
```

While visibility allows for great use cases like feature flagging, it should not be overdone. Taken to the extreme, it gets really hard to debug, test and track which "version" of the schema a user ends up seeing. Try to use visibility for broader use cases like feature flags, or different schema versions like internal vs. public.

Building a really good implementation of visibility that is smart enough to hide parts of the graph based on others, for example, where hiding a type ends up hiding fields of that type for example, can get real tricky. Consider hiding an interface, which should hide fields on types that implement it. You can also hide all fields on a type, which should hide this type, and which should, in turn, hide fields that reference it. It gets really complex! It is needed to achieve some of the feature flag and runtime masking features, but maybe not the simplest approach when it comes to building two separate GraphQL APIs. In that case, the simple approach of two definitions and code sharing is likely to work better for you.

Modular Schemas

When schemas grow large, teams often start looking for solutions to modularize or split their schemas into different parts. Tools like merge-graphql-schemas and GraphQL Modules can help do that in an opinionated way. There is no doubt modularizing is a great idea in software development in general. You'll notice however that the idea of modularizing the schema definition itself is mostly a concern coming from developers building their schemas with an **SDL first approach** since the SDL is by nature a single large definition file. While the tools linked above do a great job of allowing teams to modularize parts of an SDL and merging them into a single schema, it's not always necessary.

With a code-first approach, it's usually quite easy to modularize without any special technique or library in mind. We're dealing with **code**, so modularity should already be in mind, GraphQL or not. From what I've seen I would say developers tend to worry a bit too much about specific GraphQL modules and should instead rely on their programming language conventions to achieve modularity. Don't forget that the resulting GraphQL schema that clients consume and how it is defined are two completely different things.

As far as how you should modularize things, I'm in favor of splitting into subdomains. This means related concepts should live close to each other. Here's an example of a directory structure that generally works well:

```
+-- graphql
|   +-- orders
|   |   +-- order_type.rb
|   |   +-- invoice_type.rb
|   |   +-- ...
|   +-- products
|   |   +-- product_type.rb
|   |   +-- variant_type.rb
|   +-- schema.rb
```

Within these subdomains, you may modularize more to your liking, but I don't recommend modularizing by GraphQL "member types", at least at the top level, like this:

```
+-- graphql
|   +-- object_types
|   |   +-- order_type.rb
|   |   +-- product_type.rb
|   |   +-- ...
|   +-- interfaces
|   |   +-- ...
|   +-- input_types
|   |   +-- ...
|   +-- mutations
|   |   +-- add_product.rb
|   |   +-- delete_order.rb
|   +-- schema.rb
```

The type of schema member is rarely what we're interested in when modularizing code, we would rather separate and group concerns according to their business capabilities. Remember that you can start very simple and evolve towards this over time. It's often hard to find the right sub-domains early on, and things become more clear as we go on. Modularize when it starts to be painful and keep it simple. Most of the time, there is no need for complex solutions or frameworks.

Testing

Testing GraphQL is a challenge, and it's also something that teams just starting with GraphQL find quite difficult. The dynamic and customizable nature of GraphQL makes it really hard to have confidence in our tests. The execution engine itself, especially given the context argument most libraries provide, adds another level of unpredictability. The single most important thing we noticed about testing at GitHub is that it became much easier when we followed the advice for resolver design we just covered. Well encapsulated behaviors, as simple objects, are much easier to test and are much more predictable than a GraphQL resolver. Once your actual behaviors are tested, the rest gets a bit easier. Still, we'll want to test the interface itself, and there are a few ways to do that.

Integration Testing

Even if your business logic is well isolated and tested, testing our GraphQL layer is still important. After all, GraphQL's execution engine is quite often some sort of black box with validation and coercion possibly changing the results. This adds up to the fact that combinations of fields may also give unpredictable results (even though we try to avoid it at all cost!), among many other things. Integration tests are probably the easiest and safest type of testing we can do for a GraphQL API.

Although queries can span the entire schema, testing **per object type** is something I've seen work generally well. It's difficult to give strong recommendations on testing but usually, object integration tests can contain the following:

- Tests that work accordingly for the fields returning this object (Fetching this object through the `node` field, fetching this object through a finder field like `findProduct`, etc.)
- Tests that query for all fields on the object, to make sure we're not missing anything.
- Authorization tests, especially if authorization differs per field (More on that in Chapter 4)

With that in place for all objects, we've already got some good testing going on.

Unit Testing Resolvers

There is some value in testing individual fields, especially if they have more complex sets of parameters. Resolvers are usually simple functions, meaning they can be tested quite easily. However, there are a few things to keep in mind that may make testing individual resolvers a bit hard. First, resolvers often go through additional transformations before the field result gets added to the response payload. Type coercion, which almost all GraphQL frameworks will do for you, may change the results. There are also all the middleware and plugins you might have added to your implementation that may modify the behavior or result of your fields. Then, there's the `context` object. It's often quite hard to provide a context object to a single resolver without a global query in mind. Finally, the parent object that the resolver gets as a first argument must be mocked or provided correctly. There is no guarantee it will always be the same object when actual queries get executed.

Summary

Implementing GraphQL servers depends a lot on what your current architecture looks like, what language you're building in, and what GraphQL framework you end up picking. However, there are some ground principles that I think will help anyone trying to build a great GraphQL platform.

- Prefer code-first frameworks with high extendability (metadata, plugin and middlewares, etc)
- Keep your GraphQL layer as **thin** as possible, and refactor logic to its own domain layer if not already the case.
- Keep resolvers as simple as possible, and don't rely on global mutable state.
- Modularize when it starts hurting, you don't need a magical or specific framework. Use your programming language to achieve modularity.
- Test most of the domain logic at the domain layer. Integration tests are the best "bang for buck" approach for GraphQL servers.
- Use visibility filters for small schema variations based on runtime conditions, but don't hesitate to build completely different servers at build time when dealing with wildly different schemas.

Security

Security is a hot topic in the GraphQL world. Part of it is because people who hear about "clients can query exactly what they need" for the first time start getting anxious about clients accessing data they shouldn't have access to. As we saw, in fact, servers expose the use cases they want to expose, and nothing more. Still, there are some things to set up to make sure clients don't take down our server or access things they shouldn't. A lot of these things are similar to other API styles, but we'll cover some GraphQL specifics in this chapter.

Rate Limiting

Rate limiting is a very important security feature for any web API. While our APIs must respond to client use cases, it's also important for us to define limits within which an API client must play nicely. For endpoint-based APIs, these limits are commonly expressed in terms of requests per minute or similar. It makes sense, given the load on our servers is often related to how many endpoints are getting hit for a period of time. The problem is that it turns out this doesn't translate very well to GraphQL. To understand why, let's look at this query:

```
query A {
  me {
    name
  }
}
```

And this other request:

```
query B {
  me {
    posts(first: 100) {
      author {
        followers(first: 100) {
          name
        }
      }
    }
  }
}
```

Using the naive approach of enforcing rate limits using a requests or queries per minute approach would fail rather quickly. Although we thought that our limit was fair, given the first query, we now realize that with the same number of requests, the second query is way too expensive. The problem here is that the second query is much more complex and expensive to run than the first one, which only fetched a single object, meaning we can't allow the same number of queries per minute for both these queries. This is why GraphQL APIs must usually completely rethink how to rate limit clients. Let's cover some techniques that enable a GraphQL API to effectively rate limit clients.

Complexity Based Approach

One of the most popular techniques to rate limit GraphQL APIs is to take a complexity approach. The basic idea is this one: given a GraphQL query document, can we estimate a **cost** for it? If we could say that query B in the previous example "costs" 100 points, and that query A costs 1 point, we could use that metric instead of the number of requests to rate limit based on time. For example, our API could allow only 1000 points per minute to be run. Query A could then be executed 1000 times in a minute, but query B only 10 times. This lets us accurately represent how costly it is to execute a query on the server-side, protecting our precious resources.

Calculating Complexity

In the previous example, we computed costs in a fairly random way. In reality, we strive to come up with a cost for a query that accurately represents the server-side cost of executing it. Not only that, but a server would generally want to compute the cost or complexity of a GraphQL query **before executing it**. This allows the API to reject very expensive requests before it's too late. Some server implementations, like GraphQL Ruby, have an analyzer API that lets you do a first pass on the query document before it is executed. It is a great place to run this logic. In JavaScript land, similar tools exist like graphql-query-complexity. For those of you using implementations that don't have such an API or library, using an AST visitor or similar tool to parse and traverse the query to compute a complexity score is your best bet.

There are many ways to compute complexity and your particular implementation might influence the way you compute costs. However, an often pretty effective heuristic is to think about the complexity and to think in terms of objects rather than fields. Scalar fields are usually not very costly for a server to compute since they come from an object that has already been loaded / de-serialized. Instead, we can compute the number of object types or "nodes" that have to be fetched by the server. For example, the following query could have a cost of 2:

```
query {
  viewer { # <== Loading the viewer costs 1
    name
    bio
    bestFriend { # <== The viewer's best friend also costs 1
      name # <== but we ignore name since it's a scalar
    }
  }
}
```

The next problem we face is that list types complicate things. A single field that returns a list type may instead load `n` objects, which is hard to predict. Fortunately, if your GraphQL API is paginated, for example by using Relay's Connection specification, we can actually compute a cost using the pagination arguments:

```
query {
  # Loading the viewer costs 1
  viewer {
    # For 1 user, fetch 100 posts, costs 1
    posts(first: 100)  {
      edge {
        node {
          # For each post, load one author: 1x100, costs 1
          author {
            name
          }
        }
      }
    }
  }
}
```

Given our approach, this query would have a cost of 102 points. This is because we loaded 1 viewer object, 1 page of 100 posts, and then 100 authors. We could make the `posts` field cost 100, but here we know that our server can efficiently load 100 posts off a user. The "multiplier" effect comes from loading one author **per post**, which is usually expensive to compute for typical server implementations. You can tweak these numbers based on how expensive data fetching is for your application and access patterns.

This still leaves us with a problem for list types that are not paginated. How can we know how many items are possibly going to be loaded? First, verify this field can really not be paginated. It's very rare a list type would not be well suited for pagination. If it is impossible, chances are the list is small and you think it will remain small, which means you can assign it an average cost that makes sense for you. It doesn't need to be perfectly accurate, but express what kind of load this could produce on the server.

You can try to rate limit based on the number of objects that were **returned in the response**. For example, if a connection actually only returns 20 items, we would have a cost of 20, but if it returned 99, the cost would be 99. This is more accurate for sure, but now our rate limit algorithm is not stable anymore. A client application could think they are well within the limits, when several items get added to a collection which makes their cost increase and start getting rate limited. This is quite upsetting for a client. I think the "static" approach is a much better choice for most.

Time Based Approach

As you can see, an accurate "complexity cost" can be tricky to compute. An alternative approach is to instead rate limit by **server time**. This is inherently closer to the true "server cost", because well, we're evaluating the cost based on how much time the server took to respond! This is often done through a middleware computing the number of milliseconds elapsed between when the request was received by the application server and when the response was sent out. I really like this approach. However, it comes with similar issues as what we discussed earlier with the complexity approach: the amount of time a server takes to execute a query depends on many factors, which means for example that on a given day, a client may be totally fine, whereas during a time where servers respond slightly slower, they might get rate limited. Some may even see this as a feature: rate limit more when servers are struggling.

Compared to the complexity approach, it's a bit less easy to understand for clients. With a complexity-based approach, you can very easily express how much a query will cost. You can even do so before the end if you share your algorithm or provide tools to compute it. With time, clients basically need to try out queries and see how long they take or adapt while deploying their applications. Both solutions are still way better than simply counting the number of requests.

Exposing Rate Limits

It's often hard for clients to know whether they're within limits, or about to get rate limited if they continue. For this reason, a lot of API providers want to communicate or expose the rate limit status of a client to help the client integrate and make sure it stays within the acceptable usage of the API. The most common way to do so is through response headers. As part of every API response, we can provide clients with different headers about their current rate limit status. This example is from GitHub's REST API:

```
Status: 200 OK
```

```
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4999
X-RateLimit-Reset: 1372700873
```

`RateLimit-Limit` is the total amount of requests a user can make in a rate limit period (an hour in this case) before it will be blocked, `RateLimit-Remaining` is the actual number of requests left before this specific user is rate limited, and `RateLimit-Reset` is a Unix timestamp representing when a new period will start, meaning the remaining amount will go back to the original limit (5000 in this case).

We can apply this exact same strategy with GraphQL, except both `Limit` and `Remaining` may represent a complexity cost, or a server time amount. The header approach works great, but GitHub's GraphQL API also opts for an additional approach by including a `rateLimit` GraphQL field. Pretty meta, right?

```graphql
query {
  rateLimit {
    cost
    limit
    remaining
    resetAt
  }
  user(id: "123) {
    login
  }
}
```

Under `rateLimit`, you can query the cost of the current executed query, as well as `limit`, `remaining`, and `resetAt` just like with the headers. Here's another interesting thing: we can even ask for this information without actually executing the query by passing a `dryRun` argument:

```graphql
query {
  # Dry run means the server will
  # compute the complexity, but won't
  # fully execute the query
  rateLimit(dryRun: true) {
    cost
    limit
    remaining
    resetAt
  }
  user(id: "123) {
    login
  }
}
```

Note that response "extensions" are a great fit for this kind of information as well. For example, Shopify's API makes great use of the extensions key to provide rate limit data:

```json
{
  "data": {
    "shop": {
      "name": "ProductionReadyGraphQL"
    }
  },
  "extensions": {
    "cost": {
      "requestedQueryCost": 1,
      "actualQueryCost": 1,
      "throttleStatus": {
        "maximumAvailable": 1000,
        "currentlyAvailable": 999,
        "restoreRate": 50
      }
    }
  }
}
```

This can be very useful for clients wanting to estimate how many queries like this they could make.

Limitations Exposing rate limit details to clients is very useful, but we have to know that it comes with a few gotchas. Mainly:

- It encourages clients to “game” the system by making an exact number of requests per hour/minute to always stay under the limit. This is not necessarily a bad thing but can be very hard to change down the line.
- It assumes you can reliably and consistently provide an accurate picture of the current state of your system. If requests can be routed to different data centers, or if it depends on some information that cannot be fetched synchronously, these details can be hard to provide.

If we think these limitations could be problematic, another good approach is to document what proper usage looks like, and encourage and educate clients on how to play well with rate limits. Instead of giving them exact information on their rate limit status, we expect them to react well once they’re rate limited. A great approach to let a client know they’ve been rate limited is by using the `429 TOO MANY REQUESTS` status code, with a `Retry-After` header which lets clients know when to try again.

Blocking Abusive Queries

GraphQL’s power is all about giving a lot of power to the clients. However, we can’t just give all the power to clients. We have to set limits. One thing most people first think about is not allowing infinite depth for queries. After all, GraphQL does enable clients to build very nested queries:

```graphql
query {
  product {
    variants {
      product {
        variants {
          product {
            # We can do that for a while
            variants {}
          }
        }
      }
    }
  }
}
```

In practice, this can be implemented as a custom validator. For example, GraphQL-JS has packages available already to do depth validation, and GraphQL Ruby implements it out of the box.

You hear a lot about protecting GraphQL servers from “recursive” queries (which is not a thing the GraphQL query language allows in the first place since leaf nodes must always be selected) and limiting the depth of a query. It turns out that query depth is only one of the ways malicious clients can abuse or make excessive use of a GraphQL server. In fact, breadth can be just as bad:

```
query {
  product1:  product(id: "1") { ... }
  product2:  product(id: "1") { ... }
  product3:  product(id: "1") { ... }
  product4:  product(id: "1") { ... }
  product5:  product(id: "1") { ... }
  # ...
}
```

So how do we guard against this? The complexity approach we covered for rate limiting actually covers this pretty well. Instead of a max depth or max breadth, a **max complexity** can be set. Note that while rate limiting protects our server against abuse over time, it's a great idea to set a **maximal complexity per query** as well, so that we don't allow gigantic queries even if they are sent at a slow rate.

Besides complexity, another approach to query limits is to have a node limit. Often, this effectively translates into how many instances of an object type is requested by a query. GitHub has both a node limit per query and a complexity based rate limiting approach. Together, they ensure a client is not able to build a ridiculously large query, and stays within acceptable usage of the API over time.

Finally, no matter how we compute complexity of a query, there are often ways around them. I strongly suggest you set limits on the total byte size of the query and variables (even if they are quite large) to block excessive queries you did not even think could exist. For example, it's possible to overload certain GraphQL servers with enormous lists of arguments. You can block these by enforcing a limit on the number of items in list arguments, but the total size limit at least blocks a few more of these cases.

Timeouts

No matter what approaches we use for blocking abusive queries before executing them or rate limiting clients, chances are we may be still open to running queries that are simply too complex. Like any web server, aggressive timeouts should be set on the request time, to never let queries running for too long. There's no perfect amount as far as the timeout value is, as long as you have one.

Timeouts are interesting with GraphQL since they almost become a feature rather than an exception compared to a typical web API. Timeouts are to be expected with queries that simply request too much compute time. A high amount of timeouts for a REST endpoint is usually an emergency. People might get paged. With GraphQL, you'll most likely have to get used to them if your server accepts arbitrary queries.

With a timeout in place, we can be a bit more confident that no matter what happens with our node or complexity limits, there is an upper end of time an actor can take with a single query. The key here is finding a max complexity and/or node limit that would block queries **before** we need to timeout requests. This is easier said than done, but with good monitoring, data analysis, and some trial and error, accurate limits can be found.

Authentication

Authentication is the act of determining **who** a user is, and whether they are logged in or not. Not to be confused with **authorization**, which is the act of determining if a user **can** do an action or see a certain resource.

The main question around authentication in the GraphQL community is if it should be handled within the GraphQL server, or out of band. Practically, this is asking if our GraphQL schemas should offer `login` and `logout` style mutations, or if they should simply expect a user to be already logged in before interacting with it.

I strongly recommend leaving authentication concerns out of the GraphQL schema, and simply expect a `currentUser` or other session concepts to be present in the GraphQL context when executing a query. Resolvers should not be aware of HTTP headers or tokens. This way we can swap out, or support many different authentication schemes without changing the schema. It also makes the schema much easier to interact with, and more stateless.

Having authentication mutations on a GraphQL server either turns the server into a stateful thing, or requires us to handle authentication per-field, meaning `login` may be called with a token, but not other mutations. This is possible but I find it's more brittle than simply expecting a token to be always present when executing GraphQL queries.

This also makes authentication checks much simpler on the GraphQL side. The downside, of course, is that clients may need to authenticate using another solution than GraphQL. This somehow never felt like a big down side to me. GraphQL does not provide many advantages to a simple HTTP request when using a `login` mutation that simply fetches a single token.

I advise you use standard authentication mechanisms like an authentication middleware rather than trying to roll your own authentication in GraphQL. This might save you a few headaches!

Authorization

Authorization for APIs is a complex subject, no matter if we use GraphQL, REST or gRPC. The truth is that authorization is a hard problem in general.

It would take a whole book, and probably scientific research, to find out the ultimate way to perform authorizations. In fact there probably is no "ultimate way" and that's why recommending the best way to do it in GraphQL is not possible.

Instead, you want GraphQL to rely as much as possible on the authorization you already have deeper in your application. The main reason we want to avoid having all authorization logic at the GraphQL level is that GraphQL is often just one possible way to access your domain logic. If we start implementing everything in our GraphQL layer, we're now in a situation where we have to make sure to copy these business rules everywhere else and maintain them while they evolve, which is very error-prone.

Often, when we talk about authorization, we tend to conflate multiple concepts. On one end, we have authorization like **API scopes**; which fields or types a client is allowed to access, often on behalf of a user (OAuth for example). On the other end, we have authorization scenarios that are closer to our domain like "You can't close an issue if you're not an administrator". Generally, API scopes make a lot of sense to be implemented at the GraphQL layer, but **business rules that relate to our domain should stay as much as possible out of our GraphQL logic**.

There's no doubt that some authorization checks may need to be made at the GraphQL layer. In these cases, there are a few considerations we can keep in mind.

Prioritize Object Authorization over Field Authorization

When it's time to enforce authorization in GraphQL APIs, a very common question is whether they should be enforced on a per-type or per-field basis.

I highly recommend you start with an approach where authorization rules apply to object types rather than fields. There are a few reasons why:

First, object types usually translate quite well to API scopes. Second, the simple scalar fields on a given object typically share the same set of required permissions.

Most importantly, it's very hard to track all the possible ways to get to an object. If we only make authorization checks at field level, we open ourselves up to access patterns we didn't think of. Here's an example which uses a simple `@authorization` directive to help visualize where authorization checks would happen:

```graphql
type Query {
  adminThings: AdminOnlyType!
    @authorization(scopes: ["read:admin_only_types"])
}
```

With this simple schema, only clients with the scope `read:admin_only_types` can have access, because there's only one way to get there, through the `adminThings` field, and that field is well protected by our field authorization. Now, imagine as the schema gets more complex, and as different teams start adding their part of the schema:

```
type Query {
  adminThings: AdminOnlyType!
    @authorization(scopes: ["read:admin_only_types"])
  product: Product!
    @authorization(scopes: ["read:products"])
}

type Product {
  name: String
  settings: AdminOnlyType!
}
```

Involuntarily, we have opened a door to the `AdminOnlyType` through the `settings` field on the `Product` type, which only requires the scope `read:products`. This is still a fairly basic example, but imagine what happens when your schema grows to thousands of types. It will get very hard to track and test. The `Node` interface and field make this even more obvious. Often we can access almost everything from a Global ID, which points us towards protecting types rather than the fields that return them.

GraphQL-Ruby takes this approach by default, with authorization hooks it provides out of the box:

```ruby
class Types::Product < Types::BaseObject
  REQUIRED_SCOPE = "read:products"

  def self.authorized?(_object, context)
    context[:scopes].include?(REQUIRED_SCOPE)
  end
end
```

> Note: GraphQL ruby also allows you to make checks on the actual object returned by the parent resolver, but as we covered earlier, a lot of these checks are better made in your business layer rather than a GraphQL type definition.

GraphQL-Shield is also a popular library in JavaScript that enables you to write a complex permission layer for GraphQL. You can express a lot of permission rules with it, so do try to keep your API layer simple if you can, and be careful with the per-field checking approach. All in all, my recommendation is to keep authorization simple in your GraphQL interface. It is most probably the best thing to do for security and maintainability. Focus on API permissions like API scopes rather than business rules, and enforce them on a per-type basis at first, until you start needing finer-grained (per-field) permissions in there.

Leaking Existence

A common gotcha in API authorization is the subtle difference between: "This thing you're looking for exists but you cannot access it", and "This thing doesn't exist (it actually does but I'm not telling you)".

A lot of the time we don't want to tell someone they don't have access to a thing they tried accessing. Take for example the `node(id: ID!)` field which allows clients to fetch types that implement the Node interface using a global identifier. If clients got an error like "You can't access this object", this instantly leaks the existence of this object to the client.

To avoid that issue, we simply return `null` instead of returning an error. This means our type better be nullable, or else we risk nulling out a lot more of the response, another good reason to think twice before making a field non-nullable, as we discussed in Chapter 2 on schema design.

Blocking Introspection

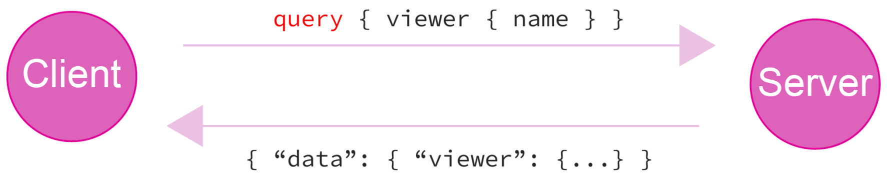
One of the most popular concerns I see around securing GraphQL APIs is removing introspection capabilities from a server. This is generally a bit odd to me since that's one of the reasons that make a GraphQL API so great to use. But of course, that depends on the context behind using that GraphQL API. Internal APIs that want to hide upcoming features, especially if they're accessible through something public like a browser might need to limit introspection to avoid leaking secret things. Internal APIs may also want to use query whitelisting, a process in which a GraphQL server only allows a **known set** of queries to be executed, often registered beforehand. This can often be used for private APIs that are used by client-side apps to block anyone from executing other queries than the ones the client makes. Introspection is before all a tool for engineers/developers, not end users. This means it should be enabled in development, but there is no need to leave it open in production for an internal API.

For public GraphQL API though, there is nothing inherently insecure about introspection since the schema is what we actually want to expose. Limiting introspection in those cases oddly sounds like "Security by obscurity". For types and fields that should not be discovered, for example feature flags as we've covered earlier in the book I prefer using schema visibility (which we've covered in Chapter 2): This allows us to hide certain parts of our schema to certain clients only.

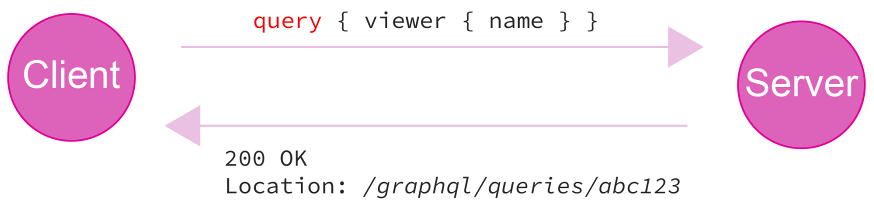
If you don't have these two things in place and they require too much effort, limiting introspection can be used as a simpler measure. However, keep in mind you might be restricting a very important feature for clients and tools.

Persisted Queries

Persisted queries are a very powerful concept that utilizes GraphQL's strength while minimizing a lot of its pain points. Let's take a look at the normal flow of a GraphQL query, which by now you probably know quite well:

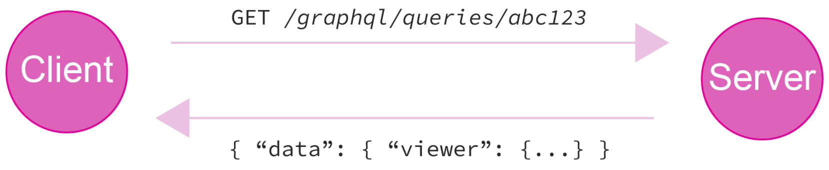


In this scenario, the client sends the typical GraphQL query to a server, the server lexes, parses, validates and executes the query, and then returns the result to the client. Let's say this client was deployed to production. One thing we notice is that the query string the client sends never changes. In fact, sending the entire query to the server every time is totally useless! We are wasting precious server side cycles by making the server parse and validate the same query string over and over again. Persisted queries attempt to solve that problem. Here's how:



With persisted queries, instead of sending the full query document on every request, the client starts by registering queries with the server, before any query is even sent. Sometimes these queries are registered before or during the deploy process. In other cases, the first query from a client is used as the "registration". In exchange, a GraphQL server capable of supporting persisted queries will provide the client with an identifier for that query. Examples of good identifiers are query hashes, URLs at which the queries can be accessed, or simple IDs.

Once the client has the identifier for a particular query, it can send the identifier along with any needed variables to execute the query, this time without passing the full query document. For example, if the server returned an URL after the registration of a certain query, the client could use this URL instead of sending the query document every single time:



This has several amazing advantages. First, clients never send the full query string anymore, saving a lot on bandwidth. But not only that, servers can optimize queries by pre-parsing them, pre-validating them, and pre-analyzing them. These things often become quite costly over time especially with large queries which makes persisted queries a very good idea for all serious GraphQL APIs.

Besides the performance and bandwidth improvements, this also helps API providers secure GraphQL APIs. Earlier we covered whitelisted queries, which meant allowing only certain queries to be run against our GraphQL server. With persisted queries, this is even more straightforward, since an API provider could allow only pre-registered queries to be run, essentially blocking access to all other queries against the API.

The funny thing with persisted queries implemented that way is that it starts to look a lot like what we wanted to escape in the first place, endpoint based and fixed queries! However, there’s a small detail that makes this so powerful. Even though we’re dealing with static queries/resources, these resources are generated by the **clients** rather than the server. In fact I love thinking of persisted queries as client dynamically generated resources, using the dynamic GraphQL engine to support as many different resources as needed by clients.

Apollo has good libraries around persisted queries and a lot of server side libraries have functionality to cache or persist queries. For example, GraphQL Java has support for caching pre-parsed queries and GraphQL-Ruby supports an operation store in its pro version.

Persisted queries are a must for all internal APIs, and I suspect they might become useful for public APIs as well eventually.

Summary

- Rate limiting GraphQL requires more thought than a typical endpoint-based API.
- A complexity or time-based approach is your best choice for rate limiting clients.
- Timeouts are a must to avoid long-running queries taking up too much server time.
- Query depth is not as important as advertised, complexity and node count is often enough.
- Authorizing object types is often simpler and less error-prone than authorizing fields.
- Disabling introspection is a good idea for private APIs, but should be avoided for public APIs.
- Persisted queries are a very powerful concept, especially for internal APIs.

Performance & Monitoring

Performance is crucial for APIs, but GraphQL makes it a bit trickier in many ways:

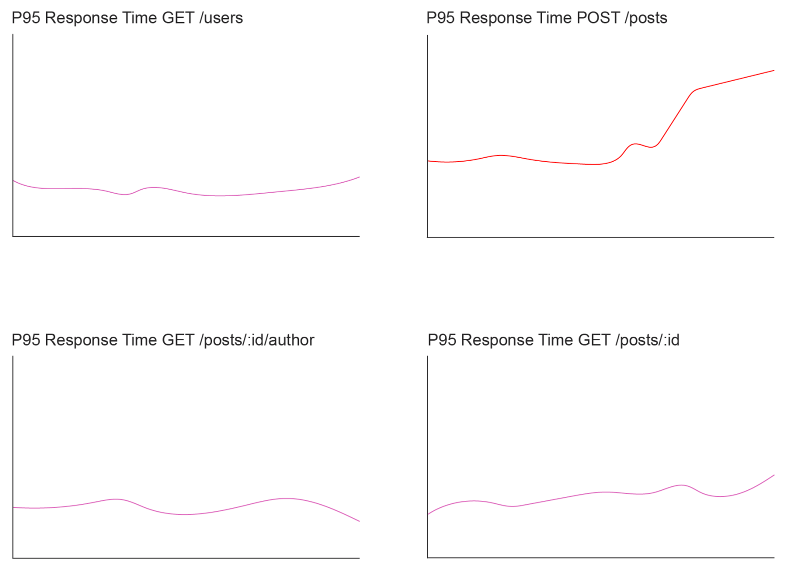
- The potential for so many different variations of requests makes it very hard to optimize for one particular query.
- The resolver pattern makes it hard for fields to cache or perform work ahead of time.
- The GraphQL execution sometimes looks like a kind of black box.

That's the tradeoff we are making when building a GraphQL server, but it doesn't mean there are no solutions to those things. In this chapter, we'll cover how to tell if a GraphQL server is performant, and how to do efficient data loading and caching.

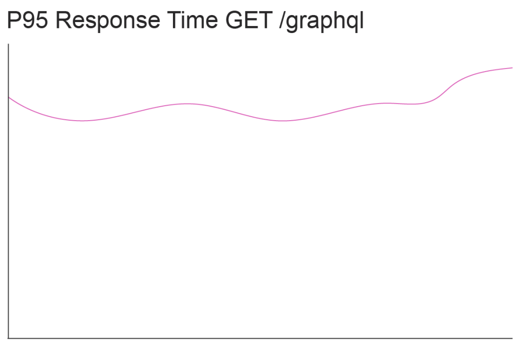
Monitoring

It's useless to work on performance if we can't even measure performance gains or detect issues with the performance of our API.

If you're used to endpoint-based HTTP APIs, you might have successfully performed some monitoring before. For example, one of the most basic ways of monitoring performance for an API is to measure response time. We may have some dashboards that look like this:



Monitoring our APIs this way lets us quickly catch a subset of the API that is having issues. For example, in the example above, something seems to be off with the POST /posts endpoint. So our first instinct might be to approach a GraphQL API the same way:



Unfortunately, we rapidly discover that measuring the response time of a GraphQL endpoint gives us almost no insight into the health of our GraphQL API. Let's see why, by looking at a super simplified example. Imagine you maintain a GraphQL API that currently mainly serves simple queries like this one:

```graphql
query {
 viewer {
  name
  bestFriend {
   name
  }
 }
}
```

A new application onboards on your API and has different, more complex, requirements:

```
query {
 viewer {
  friends(first: 1000) {
   bestFriend {
    name
    favoriteCities(first: 100) {
     name
     population
    }
   }
  }
 }
}
```

Notice how the queries look similar in terms of query document size, but how that new query is actually requesting potentially more than 100_000 objects, while the first one is only querying for one. If you were monitoring the response time of this GraphQL endpoint, and that the new client sent enough of these queries, you would see the response time for `/graphql` go up, probably by quite a bit.

Now is this alarming? Not at all, because indeed, responses we return are a bit slower because we simply serve more complex queries! Most of us are not actually interested in this, but would rather know if the performance degraded, given the same usual load/use cases. That's when a fix would be required.

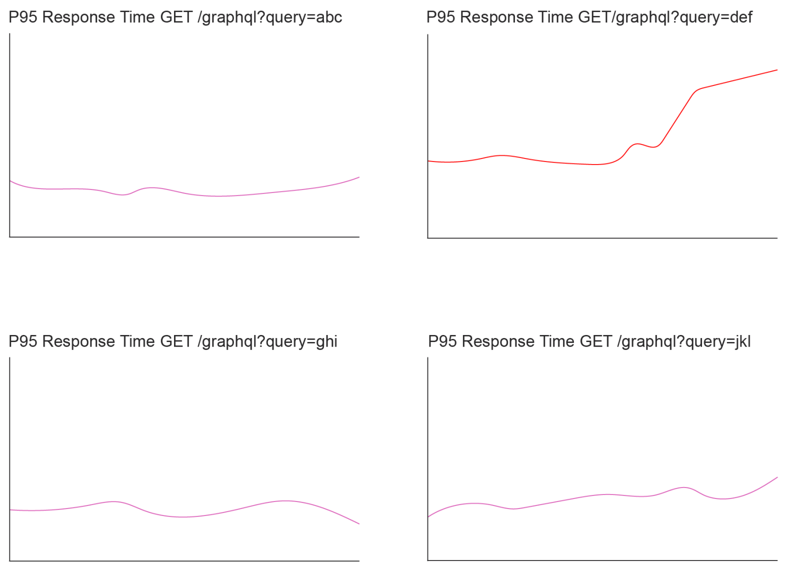
In fact, we are not interested in monitoring the endpoint, but the **queries** in this case. We want to know if a query that a user normally ran in 200ms now takes 500ms to run. If you're the maintainer of a private API, with a small set of known clients and a small set of queries, we could actually just do that — monitor known queries for their performance.

This brings us to a very important point. Your clients should always include information that helps you determine **who is making the queries**. For a public API, that's often a given if you are using authentication tokens, but I often see teams using GraphQL internally not tracking at all who their clients are. You should consider passing a client identifier and the client app's version on every call.

We could go as far as refusing to serve queries if clients don't pass this information along with queries. A common way to do so is by having clients include this information through headers:

```
GraphQL-Client-Name: my-app
GraphQL-Client-Version: 15.4.2
```

This is even easier with "persisted queries" that we saw in the last chapter. We can attach these client identifiers along with their persisted queries.



However, if you're managing a public API, or a very large internal API with a larger set of clients and queries, this may not be easy to do, due to the high cardinality of queries. We must then find other ways.

Per-Field Monitoring & Tracing

I like to think of the lifecycle of a GraphQL query in 3 broad steps:

- Parsing & Lexing
- Validation & Static Analysis
- Execution

It's often useful to monitor timings for each of those, as the execution of the query is not always where problems will arise. I've seen countless examples of a validator being incredibly slow to parse some complex queries, and even parsing that is taking up most of the computation time for a GraphQL Query. A lot of GraphQL libraries have hooks for you to insert custom tracing logic.

Per field monitoring is a much more fine-grained way of getting useful data. However, it can be costly depending on how you collect this kind of monitoring data. The idea is that on top of monitoring the whole response time of a query string, we look at the individual field performances. This way, we can detect outliers much better than when looking at global query performance.

Field and resolver monitoring can often be achieved by using middleware or resolver extensions that many libraries offer along with your favorite application performance management library/vendor. For example GraphQL-Ruby makes it really easy to instrument queries using Prometheus and many other vendors/tools. If you'd rather have something out of the box, Apollo Server along with Apollo Engine can get you metrics and logging really easily.

GraphQL is also a perfect candidate for tracing. Viewing the execution of a GraphQL query as a fine-grained trace often gives us a lot more information than the total response time for a full query. A lot of GraphQL implementation provides hooks for tracing implementations like OpenTracing.

GraphQL Response Extensions

It's often very useful to provide inline performance information in a query response, for example when debugging a slow query. The GraphQL specification allows servers to include additional information as part of the response under an `extensions` key. This is incredibly useful for metadata like tracing information. Apollo Tracing defines a tracing extension format; however, it is not necessarily an official standard. You may come up with your own tracing extension as well.

```
// An example of the Apollo Tracing extension
{
  "data": <>,
  "errors": <>,
  "extensions": {
    "tracing": {
      "version": 1,
      "startTime": <>,
      "endTime": <>,
      "duration": <>,
      "parsing": {
        "startOffset": <>,
        "duration": <>,
      },
      "validation": {
        "startOffset": <>,
        "duration": <>,
      },
      "execution": {
        "resolvers": [
          {
            "path": [<>, ...],
            "parentType": <>,
            "fieldName": <>,
            "returnType": <>,
            "startOffset": <>,
            "duration": <>,
          },
          ...
        ]
      }
    }
  }
}
```

A very important thing to think about is not only encoding the timings of each resolver, but also the timings of every external call that resolvers make. It is very rare for a resolver itself to be slow in terms of CPU. The huge majority of performance issues are from resolvers making external calls, like calls to a cache, a database, or an external service. These are extremely useful to include in traces like this. You seldom want to return this trace for all responses, unless you expect all clients to use this information because it often adds a lot of bytes to the response. At the very least you should make sure it is compressed. If you maintain a public API, you might not want to expose such a detailed trace for security.

Slow Query Log

Slow query logs are an idea that comes from database implementations. The idea is simple: set a threshold at which we consider a query too slow. Any query that exceed that threshold get logged. This is most useful when dealing with a known set of queries, or when dealing with either a public or a large enough set of clients and queries. We then get the same problem described at the beginning of the chapter: are the slow queries problematic or simply slow because they are larger queries? Still, the slow query log can be a simple and effective tool to catch expensive queries early on, before they become problematic.

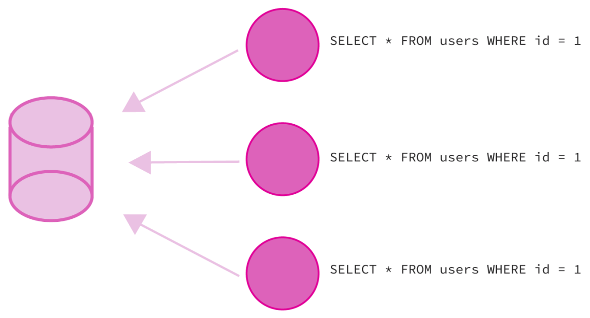
Tracking Queries over Time

Regressions are often what we are really interested in, in terms of performance. Are there any queries that have gotten slower in the last hour, day, week? If you can afford it, tracking every single query over time, in something like a Time Series database or your Data Warehouse, can be very useful. Products like Apollo Platform give you similar features if you don't want to build it yourself. Because query strings can vary a lot depending on white space, arguments, and order of fields, a hash or signature of a query often needs to be computed. We can then track the performance of a particular query hash instead of tracking plain query strings. GitHub computes a hash for both query strings and the variables provided. We can then track regressions for a particular pair of query string + variables, avoiding the monitoring issue we talked about earlier (loading 250 items is often inherently slower than loading 1).

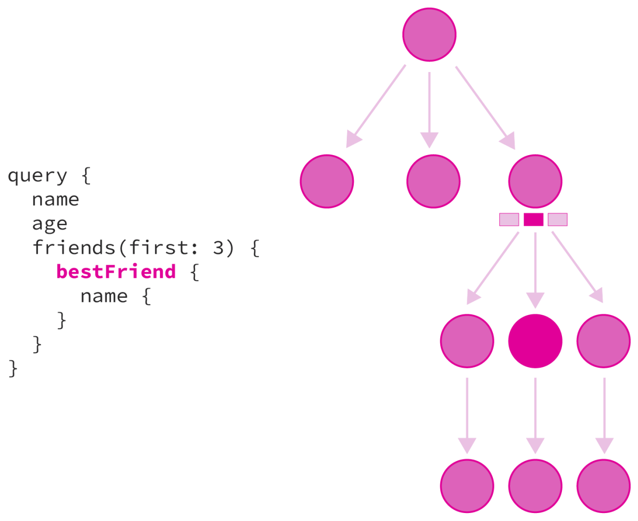
The N+1 Problem and the Dataloader Pattern

While a very powerful concept that allows a GraphQL engine to dynamically generate client representations, the "resolver" pattern – the functions that most GraphQL implementations use to execute queries – can sometimes lead to certain unexpected issues when used naively.

This is the case with data loading. The problem is that resolvers live in their own little world (in fact, they can even be executed in parallel along others). This means that a resolver with data requirements has no idea if this data has been loaded before, or if it will be loaded after. For example, three resolvers that need to load a certain user could end up making the same SQL query:

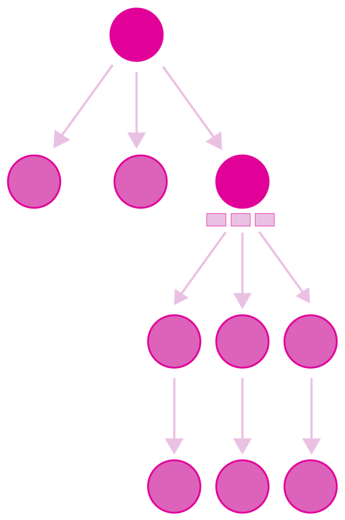

SELECT * FROM users WHERE id = 1
SELECT * FROM users WHERE id = 1
SELECT * FROM users WHERE id = 1

Most servers will actually resolve queries serially, meaning one field after the other. Take a look at the execution of a typical query that loads the current user's name and age, all their friends, as well as the best friend of each of their friends.



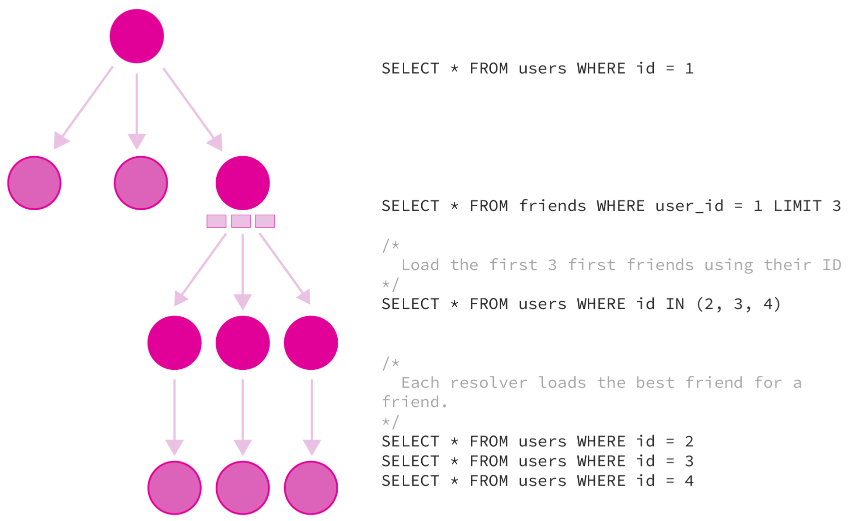
```
query {
  name
  age
  friends(first: 3) {
    bestFriend {
      name {
    }
  }
}
```

Looking at this GraphQL query from an external eye, we can imagine how we would want to load the data that we need. In fact, if we were dealing with an endpoint-based API, where the serialization logic for the whole payload is often collocated or at least executed in a shared context, we'd know how to do this. Here, we want to load the current user, load the first 3 friends from a join table, and then load these 3 friends at once using their IDs. Finally, we would take the `best_friend_id` from each user and load them all. Looking at a resolver graph, we'd have the 4 following queries:



```
SELECT * FROM users WHERE id = 1

/*
  Load the first 3 first friends IDs
*/
SELECT * FROM friends WHERE user_id = 1 LIMIT 3

/*
  Load the first 3 first friends using their ID
*/
SELECT * FROM users WHERE id IN (2, 3, 4)

/*
  Load the best friend of each
*/
SELECT * FROM users WHERE id IN (5, 6, 7)
```

As we just saw, this is hard to achieve with our resolver concept. How can the resolver for `friends(first: 3)` know that it needs to preload the best friend for each? This is the responsibility of the `bestFriend` field! While the fact that the loading of `bestFriend` is collocated in the `bestFriend` resolver is great, a naive execution of this query would rather look like this:



```
SELECT * FROM users WHERE id = 1


SELECT * FROM friends WHERE user_id = 1 LIMIT 3

/*
  Load the first 3 first friends using their ID
*/
SELECT * FROM users WHERE id IN (2, 3, 4)


/*
  Each resolver loads the best friend for a
friend.
*/
SELECT * FROM users WHERE id = 2
SELECT * FROM users WHERE id = 3
SELECT * FROM users WHERE id = 4
```

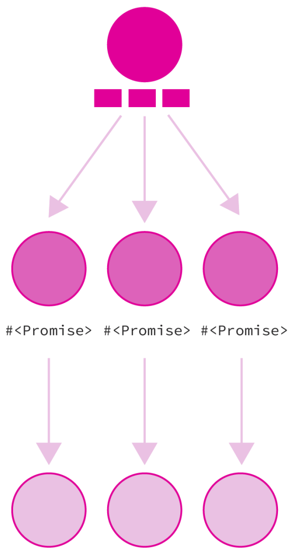
As you can see, a typical GraphQL execution would likely make 6 queries instead of 4 here. Ask for 50 users and we would have 53 queries while the other solution still works with 4 SQL calls! This is clearly not acceptable and will fall apart very quickly as your data sets grow and more queries of that kind are run against your GraphQL API. Now that we see the problem, what can we do about this? There are multiple ways to look at the problem. The first one is to ask ourselves if we could not find a way to load data ahead of time, instead of waiting for child resolvers to load their small part of data. In this case, this could mean for the `friends` resolver to "look ahead" and see that the best friend will need to be loaded for each. It could then preload this data and each bestFriend resolver could simply use a part of this preloaded data.

This solution is not the most popular one, and that's understandable. A GraphQL server will usually let clients query data in the representation they like. This means our loading system would need to adapt to every single scenario of data requirements that could appear very far into a query. It is definitely doable, but from what I've seen so far, most solutions out there are quite naive and will eventually break in very complex data loading scenarios. Instead, the more popular approach at the moment is one that is commonly called "DataLoader". This is because the first implementation of this pattern for GraphQL was released as a JavaScript library called `DataLoader`.

Lazy Loading

The idea behind DataLoader is kind of the opposite of the "look-ahead" solution we just talked about. Instead of being eager about loading data, and having to handle all possible cases ahead, the DataLoader style loading pattern is purposefully very lazy about loading data. Let's dive into it.

The first principle to understand is that when using the DataLoader approach, we take an asynchronous approach to resolvers. This means resolvers don't always return a value anymore, they can return somewhat of an "incomplete result". For the sake of this example, we'll talk about the most commonly used "incomplete result" objects, promises.



As you can see, while the standard execution strategy would execute the query in a depth-first-search approach, resolving child fields before other fields at the same level, we have a different approach here. When a resolver wants to fetch data, instead of fetching it right away, it will indicate to the executor that it will eventually have data, but that for now, it should proceed to the next resolver on the same level of the query tree.

The next step to understand how this eventually works is to introduce the concept of loaders. Loaders are a simple idea, even though their implementation may be complex. The basic idea is that when an individual resolver has data needs, it will go through a loader instead of going straight to a data store. The role of loaders is to collect identifiers required to fetch objects from individual resolvers and to batch-load this data more efficiently.

The typical abstraction for a loader is a class or object with two main methods:

- `#load` takes the loading key as an argument for the data the caller is interested in and returns a promise, which will eventually be fulfilled with the data that the caller asked for. This method is used within resolvers.

- `#perform` (batchFunction) takes all the accumulated keys that the load function calls added, and loads the data in the most efficient way. This method is usually either defined by us, or calls a batch function we've provided.

```
class Loader {
 load(key) {
  // Adds the key to an eventual batch and returns a promise
 }

 perform(keys) {
  // Receives all keys that were asked to be loaded
  // Loads them all as a batch
  // Fulfills every resolver promise with the
  // data they've asked for
 }
}
```

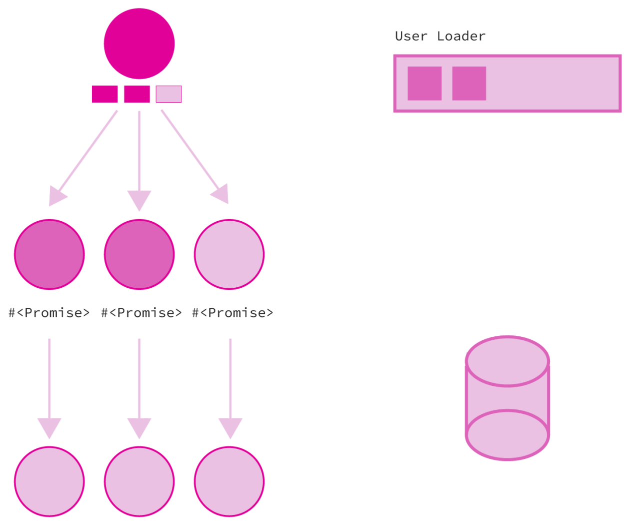
Here's an actual example using the Dataloader package and GraphQL-Js:

```
// Create a loader that can fetch multiple users
// in a single batch
const userLoader = new DataLoader(ids => getUsers(ids));

const UserType = new GraphQLObjectType({
 name: 'User',
 fields: () => ({
  name: { type: GraphQLString },
  bestFriend: {
   type: UserType,
   // The bestFriend resolver now returns a Promise
   // instead of loading the user right away.
   resolve: user => userLoader.load(user.bestFriendID)
  },
 })
})
```

One thing that is usually hard to grasp at first is the time when that perform or batch function is called during the execution. We know resolvers won't load data individually anymore, but when are these promises even fulfilled? How does the GraphQL execution engine handle this? It turns out that it varies a lot, depending on the implementation and language.

For example, Node.js has asynchronous primitives that allow this pattern to work quite well. This is why DataLoader uses process.nextTick to batch-load a set of keys. DataLoader uses Node.js queue system to wait for all promises to have been enqueued and then run the batch function. If you would like more details into how this works under the hood, Lee Byron has an amazing video explanation available. I highly recommend watching this video, at the very least the part about enqueuePostPromiseJob.

Other languages do it differently. For example, GraphQL-Ruby has a specific "lazy executor" that starts by resolving all field resolver functions at a single level of a query, until it can't go any further without resolving promises. In that case, it calls the batch functions on loaders, and keeps going on with the execution once the promises are fulfilled.



If you're looking into implementing DataLoader, the JavaScript implementation repository lists out a lot of language-specific implementations of this pattern that you can pick from.

Lazy Loading Drawbacks

Lazy loading is often a very important piece when it comes to the performance of a GraphQL server, but it has a few drawbacks.

- Monitoring gets a bit harder; the execution is not only about individual resolver timings anymore.
- The execution mental model is not as clear as before, making it hard to debug performance issues.
- The performance of an individual field can be a lie. Imagine a field that

simply enqueues thousands of users to be loaded. If we were to monitor the performance of that field, we'd see that it resolves very fast. In fact, it's putting all of its work on a loader, which we have to monitor separately.

- Everything becomes async! If you're used to it, especially working with JavaScript, that might be fine, but if you're working in a language with poor support for promises/futures, this may be quite annoying to work with.

Caching

If you've followed the discussions around whether GraphQL is a good idea or not, you might have heard things like "GraphQL breaks caching", or "GraphQL is not cacheable". If not, I guarantee you'll be hearing similar things when you start displaying interest in building a GraphQL API. I've seen some companies starting to use GraphQL being scared of this question to which they don't have a clear answer. Before we dive into the world of caching and GraphQL, it might be a good idea to address these common concerns and understand where they come from.

Comments like "GraphQL breaks caching" lack the nuance required to actually have a proper discussion about caching and GraphQL. What kind of caching? Client side? Server side? HTTP caching? Application side caching? To have a proper discussion and end up with a better understanding of GraphQL's limitations in terms of caching, we must take certain subtleties into account.

GraphQL breaks server-side caching?

This is a common thing to see thrown around when talking about GraphQL. The first thing to understand is that "server-side caching" is already vague. At this point, we know that GraphQL can actually be a thin layer over our existing servers, and that in no way does GraphQL prevent us to cache on the server-side, sometimes referred to as Application Caching. We will dive deeper into some concepts that can be applied at the application level later on in this chapter.

Most GraphQL clients and frameworks have as a feature a de-normalized cache that allows client-side applications to avoid re-fetching data that they already possess, using it to optimistically update a UI and to keep a consistent version of the world across components. So if we can actually cache things both at the server and client layers, why are we hearing so much about GraphQL "breaking", or making caching really hard? This is where it becomes more subtle.

HTTP Caching

While certain API styles like REST make great use of the powerful HTTP semantics, GraphQL does not really, at least not by default. Since GraphQL is transport agnostic, most server implementations out there use HTTP as a "dumb pipe", rather than using it to its full potential. This causes issues around certain things, like HTTP caching. There are multiple parts to HTTP caching that are important to understand before we go further.

First, there are many different cache entities that can be involved in HTTP caching. Client-side caches, such as browser caches, use HTTP caching to avoid re-fetching data that is still fresh. Gateway caches are usually deployed along with a server, to avoid requests from always hitting servers if the information is still up to date at the cache level.

Two concepts that are particularly important to understand when it comes to HTTP caching: freshness and validation. Freshness lets the server transmit, through `Cache-Control` and `Expires` HTTP headers, the time for which a resource should be considered fresh. For example, a server returning this `Cache-Control` header is telling clients not to bother fetching this resource again until it has been at least one hour (3600 seconds):

```
Cache-Control: max-age=3600
```

This is especially great for data that doesn't change often, such as browser assets. Whenever the age of the resource we fetched will be greater than this `max-age`, the client will emit a request instead of using the value in its cache. However, it doesn't mean that it actually changed on the server. This is where validation comes in. Validation is a way for clients to avoid re-fetching data when they're not sure if the data is still fresh or not. There are two common HTTP headers to achieve this. The first one is Last-Modified. When an HTTP cache on the server has a value for Last-Modified, a client can send an `If-Modified-Since` to avoid downloading the data if the data hasn't changed since last time it downloaded it.

The other common way of validating caches is using ETag. ETags are server-generated identifiers for representations that change whenever the resource has changed as well. This lets the client track which "version" of the representation it has and avoids re-downloading a representation for which the ETag is the same as the one the client has.

Together, freshness and validation are a powerful way to control client and gateway caches. Can we really not harness the power of HTTP caching with GraphQL?

GraphQL & HTTP Caching

When we dig deeper into the issues with GraphQL and Caching, we discover some of these issues are purely related to HTTP Caching. It is an important distinction to make since server-side caching could just as well mean an HTTP Gateway cache or an application-side caching on the server.

One of the first things that could influence how HTTP caching works with GraphQL is the HTTP verb that is used to send GraphQL queries. A lot of misinformation has led to some people believing that using POST on a GraphQL endpoint is the only way to make it work. HTTP caches will not cache POST requests, which means GraphQL is simply not cacheable at the HTTP level. However, GET is indeed a valid way to query a GraphQL server over HTTP. This means that caches could indeed cache GraphQL responses.

The only issue with `GET` is with the size of the query string. For example, almost every browser has different limits for these. If this becomes an issue, persisted queries become very useful. As we saw in the last chapter, persisted queries let you store query strings on the server instead of the client, meaning a client could execute queries simply like this:

```
GET /graphql/my_query
```

Our GraphQL query, with persisted queries, is essentially a typical HTTP endpoint! As we covered when talking about persisted queries, if we see GraphQL queries as a way to dynamically create server-side client-specific representations, each query is in fact something that could be cached. Can we apply HTTP concepts to GraphQL queries? Let's start with freshness. What we would want is for a server to be able to tell a client how long the query can be considered as fresh, and when to request for this data again. The unfortunate thing here is that HTTP semantics operate on whole responses/representations, and doesn't care or understand GraphQL queries, meaning we don't have a way to do per-field freshness, for example. Still, nothing could stop us from adding a freshness header to a whole query: we could say that a GraphQL query's max-age is equal to the field in the query with the lowest max-age. Validation is similar. While we can't use HTTP to revalidate only parts of the query, we could set Last-Modified to the value of the field with oldest Last-Modified value, and we could also generate an ETag based on a combination of all data loaded within the query.

Since GraphQL queries possibly span multiple entities that could change, and that they need to be represented as one representation on the GraphQL side, the number of invalidations for GraphQL queries is usually quite high. A single field being invalidated can invalidate an entire query, even if the rest of it is still "fresh".

While this may sound to some as GraphQL being inherently "not cacheable", it is instead really all about customization vs. optimization. The invalidation issue we discussed above is not something very specific to GraphQL and is something that all APIs that are dynamic and customizable as GraphQL struggle with. It's a tradeoff we choose to make!

Take for example a typical HTTP endpoint for a web API:

```
GET /user/1
```

This particular endpoint accepts no particular query parameters and simply returns the user associated to this URI. As a public API especially, this endpoint is highly cacheable across all API clients. Now imagine a more customizable version of this endpoint:

```
GET /user/1?partial=complete
GET /user/1?partial=compact
```

This API uses a partial query parameter to change the level of detail of the response. An even more customizable API, just as we saw in the introduction could look like this:

```
GET /user/1?fields=name,friends
```

The more versions of an HTTP endpoint we have, the more we dilute the cache. Meaning someone requesting fields=name only can't actually use a cache, even though someone requested `fields=name,friends`. We've got the same issue happening with GraphQL, remove a field, change anything to a query in fact, and we lose the benefit of all queries that were cached with a superset or subset of the data.

As you see, this is not something specific to GraphQL, and can be found in any API over HTTP that decides to opt for a more customizable API. With endpoint based APIs, the API designer is in charge of building the API and making these tradeoffs. By choosing GraphQL, we implicitly take the customizability road. Hopefully, that tradeoff was deliberate and the cache invalidation issues were worth it on the long run. Instead of "GraphQL is not cacheable", how about "Highly customizable APIs benefit less from HTTP caching"?

How Important is HTTP Caching to you?

There's no doubt HTTP caching is a wonderful mechanism for data that doesn't change often. It can be shared across multiple users, especially when gateway caches are concerned. For authenticated web APIs, the eternal debate lies on how useful HTTP caching really is. It is a debate which I won't solve here, but it is still worth discussing.

An interesting fact is that shared caches actually should not cache any request with an Authorization header. If your API is authenticated, the "GraphQL breaks HTTP network/gateway/shared caches" argument simply does not apply. Private caches, such as browser caches and client side caches, could still gain a lot from using HTTP caching. As we saw, it is not out of question with GraphQL, it is simply not as powerful as for highly optimized/one-size-fits-all APIs because of how often a query can be invalidated and how little can be shared.

Another thing to keep in mind is that many web APIs actually can't have stale data for very long, which means that freshness headers become less useful.

Validators such as ETag and Last-Modified usually require the server to retrieve all necessary data and run business logic to be computed. This usually is the major part of the work, savings being mainly on serialization and bandwidth since no data needs to be transmitted. If bandwidth or serialization is an issue, again, nothing prevents you to implement ETag or Last-Modified generation for a GraphQL query. GraphQL definitely made tradeoffs where it is much more suited to authenticated APIs and realtime data that changes often, versus serving long-lived data as a public API. If your use case is the latter one, and it is the only thing your API does, considering using an API architecture that uses HTTP in a more meaningful way could be a better choice.

HTTP Caching could benefit GraphQL in good ways. The lack of GraphQL over HTTP specification is something that makes things a bit harder. The fact mutations can possibly be executed using the `GET` method is an example of something that could be solved by such specification. However, there are many other ways to cache GraphQL, be it at the client level, the whole response level, the individual resolver level, etc. In fact, certain vendors, like Apollo, are exploring caching semantics right into GraphQL. In this chapter, we will mainly cover GraphQL specific approaches, since these are currently the most used tools and can be more powerful in the long run because they understand GraphQL semantics.

Caching in Practice

As you can see, caching GraphQL is a nuanced issue. While it may be less effective than with highly specific, public endpoint-based APIs, there is still a lot of value to be had by caching in GraphQL, and that, at many different levels.

Full Query Caching Most of the caching for GraphQL will be more effective at the application level, meaning within your server. The most common issue is that since GraphQL queries can span multiple entities at once, queries may sometimes use parts of the schema that should be cached and other parts that should not be cached at the same time.

For example, Shopify took a very pragmatic approach to solve this issue. Instead of trying to solve caching and GraphQL at a global level, they decided to cache queries they could cache. Some of their types represent objects that are cacheable: things on the storefront like products, variants, images, etc. They annotate the types that are cacheable when defining them. It might look like this:

```graphql
type Product @cacheable {
  name: String
}
```

When queries are executed, the server looks at all the fields and verifies **all types are cacheable**. If that's the case, the whole query is safe to be cached and is cached under a key they generate based on the query and user context. This is such a good example of a great progressive improvement since a lot of queries did hit only cacheable storefront types. Of course, the gotcha is that if a single non-cacheable field gets added, we lose all benefits of caching.

Cache Keys The generation of a cache key is always important, but even more so with GraphQL. The dynamic nature of GraphQL queries is such that even a white space in the query could affect the key and cause a miss, even though it was the same query in the first place. A good cache key should generally contain at least:

- User information (if authenticated API).
- A query hash, which should be normalized as much as possible.
- The variables hash (we would not want queries with different variables to be cached as the same thing).
- The operation name
- A cache-busting element.

The user information part is usually a `user_id` or some sort of client identifier, in order to avoid serving data that belongs to one user to all other users. The query hash is a representation of the query string the client is asking to be executed. Generally, it should be normalized to remove variations coming from things like white space, comments, etc. Some would even say field ordering should be normalized, but that can be potentially scary since the spec does say something about ordering. The variables hash is important since not all variables will always be included in the query string.

The operation name is also easy to forget but very important. In GraphQL, clients are allowed to define multiple operations within one query string:

```
query A {
  shop {
    name
  }
}

query B {
  shop {
    products {
      name
    }
  }
}
```

Although most servers do not execute multiple queries, they allow clients to provide an `operation_name`, which tells which of operations A or B it should execute. If we cached full responses without caching the `operation_name`, this would break quickly as clients that provide B as an operation name could get a result for query A, and vice-versa.

Finally, the "cache-busting element" will depend a lot on your own implementation. In the previous Shopify example, a "shop_version" attribute, which represents the current state of a shop's storefront, is used. By including it in the cache key, they ensure queries are never stale. If you can tolerate some staleness, you can also use a Time-To-Live (TTL) approach by setting an expiration on your cache key instead.

Data Layer Caching Remember the batch loaders we covered while talking about the N+1 problem? Well not only do they avoid inefficient database queries, but they can also help to cache duplicate queries. Imagine the following query:

```
query {
  shop {
    owner {
      shop {
        owner {
          shop {
            owner {
              name
            }
          }
        }
      }
    }
  }
}
```

Without caching, as we saw in the chapter on performance, we probably would be making a database query every time we hit the fields `shop` and `owner`. Fortunately, most DataLoader implementations also take care of that for you. They will not only batch calls, but cache any data loading access given a certain entity!

Data caching is probably the most effective and most essential layer you should be working on, before moving up to more advanced caching. It's much easier to batch and cache specific data sources than to try caching highly dynamic actual GraphQL queries.

Resolver Caching Full response caching can be tricky to get right because of how dynamic GraphQL queries are. For this reason, some of us may turn to caching **individual** field resolvers. I suggest you use your best judgment when trying to cache individual resolvers. While it may sound as easy as caching, in terms of field names and argument names, we often forget about the mighty **context argument** that most implementations support in resolvers. This means that introducing a generic caching solution for resolvers needs to take into account any data that could be used from that context to generate different values.

For this reason, I suggest instead that you take a look at caching individual resolvers, just as you would any other logic in your system. For example, if your resolver is making an expensive call, consider caching that field's logic specifically if it is causing problems.

HTTP Caching As we mentioned in the introduction of this section. even HTTP caching is doable in GraphQL. A good example of this is the Apollo Server. It has a solid caching implementation that can help to generate HTTP caching headers based on your schema and even using special functions in resolvers at runtime. As we said earlier, while caching might be a bit less effective with GraphQL, it is possible and can be done in a pragmatic way.

Caching Summary

Caching GraphQL might be a bit less effective due to how client-driven it is, but it doesn't mean it is impossible. Techniques like persisted queries make HTTP caching actually quite powerful with GraphQL as well. If your API consists of mainly static & public data, maybe there are more effective API styles. However, if you support multiple clients with different needs in an authenticated API with interactive data, GraphQL is a great choice and caching can be used in a pragmatic way.

Compiled Queries

Compiled queries are a very exciting area of GraphQL that I hope to see a lot more of in the future. Without going into details, compiled queries take the idea of persisted queries even further. Standard persisted queries execute the registered queries the same way as any other queries (besides skipping validation/analysis). This means that even though we know exactly what queries will be run, we still have the overhead of the GraphQL execution engine. What if we optimized that ahead of time? That's exactly what compiled queries try to achieve.

At the moment these are more at the idea stage, but there are some examples out there. GraphQL compilers and engines, new ways to execute queries are a very exciting area and they have the potential to solve a lot of the performance downsides of GraphQL.

Summary

- GraphQL is inherently harder to optimize than typical endpoint-based APIs.
- Monitoring GraphQL often requires monitoring individual fields or queries rather than monitoring endpoint response time.
- The N+1 problem can be avoided through lazy loading.
- Caching is possible with GraphQL, but often not as powerful as with other API styles.

- Compiled queries are a promising technique for improving performance of GraphQL APIs.

Tooling

I attribute a lot of the success GraphQL has had over the past few years to the amazing tooling ecosystem of GraphQL, and the amazing tooling that can be built because of its type system and specification. In this chapter, we'll cover what I think are the must-have tools for any serious GraphQL platform that needs to scale. There are so many things to be built and the truth is that the open-source and even vendor offerings are lacking, with Apollo being the major player of developer tooling in the GraphQL space. In this chapter, we'll explore some of my favorite tools to ensure a high-quality experience for GraphQL API development.

Linting

Linters are something most of us have used at least once in our lives. If you're in the JavaScript world, you might be familiar with ESLint, which is almost ubiquitous by now. We can apply the same tooling to our GraphQL development experience. The GraphQL type system and its introspection capabilities allow us to read the GraphQL definition and also analyze it.

The bigger your schema becomes, and also the more the number of contributors grows, keeping consistency across your API becomes more and more challenging. All the good design practices we saw in Chapter 2 become hard to enforce and often, a specific GraphQL team becomes a gate-keeping team, something that simply cannot scale in a larger organization. Linters allow us to encode these practices into a set of rules and have those rules run in an automated way whenever changes are made to the schema. These linters can be run locally, in development, but also in a much more powerful way during your automated tests or CI pipeline.

In a few conferences in 2018, I presented some of the tooling we built at GitHub to make sure our schema stayed high quality and consistent with more than 300 engineers actively contributing to it. One of these tools was a linter, called **GraphQL Doctor**. Since then, a few people played around with the idea. I even think there is an open-source GraphQL doctor out there. We built the linter and applied it where most of the engineers would notice: GitHub itself. Our linter ended up being a pull request bot capable of analyzing GraphQL changes and recommending best practices when needed:

There are some of these tools already out there if you don't want to build them yourself. A great example is graphql-schema-linter, a JavaScript GraphQL Schema linter.

Change Management

Not only does a linter help with quality and consistency, but it can also help us avoid breaking integrators by helping developers working on the schema to be aware of potentially dangerous changes.

The same tool we use for linting also compares schema versions and determines what changed between them. Each change is then analyzed and GraphQL doctor can then tell if these changes are breaking or not. Getting a list of differences between schemas can be annoying but fortunately, there are a few tools out there to help you:

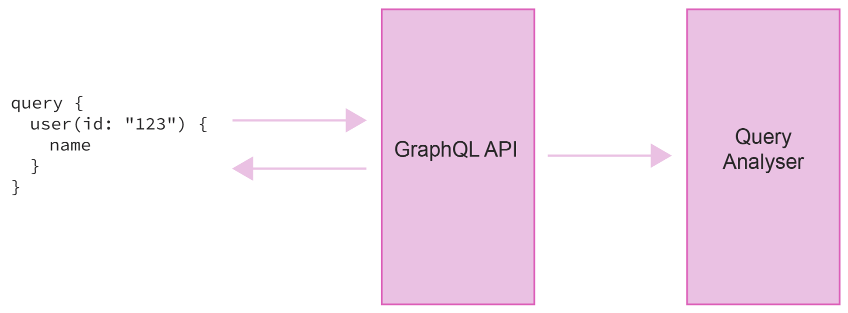
- GraphQL Schema Comparator is a tool I wrote, which GraphQL Doctor uses internally.
- GraphQL-JS includes a schema comparator, although it's a bit hidden.
- Sangria, the Scala implementation of GraphQL also contains a schema comparator.
- Finally, Apollo's Platform can help you detect these changes.

A lot of these tools help you with categorizing breaking versus non-breaking changes. It's incredibly important to know whether changes are going to be affecting clients in any way. When your team grows, it's hard to watch for these manually and tools like this can greatly help.

Analytics

An often overlooked feature of GraphQL is the fact there is no way for a client to ask for all fields on a certain type, a sort of `SELECT *`. This may sound like something against GraphQL, but it actually allows us to build amazing analytics and usage analysis. Take your typical endpoint-based API: we can track which resources are being consumed, the status codes they return, and any headers that are being sent of returned ???. However, the only thing we know is which resource a client was interested in, not which properties they are actually using. If I'm deprecating a property called `address` on a resource `/user/:id`, we have to assume we are breaking everyone, since we have no information on how the resource is exactly used. With GraphQL, we know if `address` on a `User` type has been selected, which is very powerful when it comes to understanding how our API is being used.

For APIs with various different clients, especially public APIs, tracking usage down to every field and argument can turn out to be incredibly useful. At GitHub, we took every request hitting our public API and analyzed it against the current version of the schema. Because queries can be quite large and going through every field to collect usage information can be expensive, a good idea is to handle this process outside of the process that is executing the query. This prevents altering query time for clients.



```
query {
  user(id: "123") {
    name
  }
}
```

For every query to analyze, a good idea is to include anything that could help us down the line:

- The actor behind this query: this could be an access token, the current user, the current application, etc.
- Any errors that may have happened, including parsing errors, and GraphQL errors, like if certain fields did not exist.
- Resolver timings, full query time, everything we talked about in the performance chapter.

Another issue you might hit is that queries are not valid against all versions of your schema, and we certainly don't expect the schema to stay the same. This is why it's a good idea to send, along with information about the query, information about the version of the schema that executed this query. Instead of sending the whole schema SDL to the analysis service every time, we can compute a hash of the schema, like `sha256`, and send over that hash to the analysis service.

When the analysis service receives a new query, it first needs to fetch the schema, which you can achieve in different ways. For consistency purposes, we used git to load the schema SDL, which is always stored in the API repository, which means we also included the git `sha` along with the schema hash. Once you have the schema, analysis becomes easier. Starting from the query root, we look at every field in the query, validate them, and analyze them including any metadata (Is the field deprecated? Is it accessible under a feature flag only?, etc.), and store them under a format that can allow us to search later on.

One way to do this is to "de-normalize" the whole query into a set of entities that were used:

- A list of fields (with their parent types)
- A list of arguments used (with their parent field and type)
- A list of fragment spreads that were used (very useful when finding out what concrete type users are querying against for interfaces and union types)
- Enum values used

Store this in any time series database or data warehouse you have access to. Along with all these entities, you can encode who made the query, which enables searches of that kind:

- Finding the top 10 list of integrators querying User.name
- Finding fields frequently used together
- Finding the slowest fields for a particular integrator
- Looking at the usage for all deprecated fields
- Finding if it is safe to remove an optional argument
- The sky is the limit! Add any other contextual data that might help debug and/or understand how your API is being used.

Removing sensitive parameters

It is common practice to remove sensitive parameters and user data out of queries before storing and analyzing them. There are a lot of parameter blacklists out there for HTTP requests, but for GraphQL we have to look within the query string. A good trick is to keep the query valid and to replace any user sensitive data by placeholders. Given this query:

```
mutation {
  createUser(
    name: "John",
    age: 30,
    profession: "Engineer"
  ) {
    name
    age
  }
}
```

We can replace all user-provided data with placeholder values, or simply `null` if the field is nullable:

```graphql
mutation {
  createUser(
    name: "REDACTED",
    age: null,
    profession: "REDACTED"
  ) {
    name
    age
  }
}
```

This way the query remains valid against the schema. We can analyze which arguments and fields were used, but without viewing the actual user provided values. Don't forget to sanitize parameters within list and input types as well.

As you can see, building such an analyzer is not necessarily an easy task, but it can be very useful. There are unfortunately not a lot of tools that will do that out of the box for you, since it depends so much on your architecture. Once again Apollo is your friend if you are in need of something of that kind.

Summary

GraphQL's schema enables us to use so many great tools to ensure we understand and keep our schema stable and well designed. We must take the opportunity. Use linters to enforce consistency and best practices, breaking change detectors to make sure your API stays stable, and finally, gather as much data as you can on queries. The ability to understand queries and which parts of the API they hit is such a superpower that has saved us many times in the past.

Workflow

Keeping an API high quality as teams grow is a challenge and is rarely achieved without thinking about the whole process from beginning to end. In this chapter, we'll cover important parts of a workflow that will hopefully lead you to develop one that works for your team or organization. Every team works differently, so this is far from an absolute recommendation, but these are practices that I've seen work well over the years.

Design

As we've covered many times in this book so far, careful design goes a long way in ensuring that an API stands the test of time, remains stable, easy to understand and to use by clients. If there's only one thing you should remember in this chapter, this is the one I absolutely would recommend to any team working on a GraphQL API. Too much upfront design can be less than ideal, of course. But from my experience, most teams start working on implementing API features way too early. More upfront design, at least a few discussions on the subject, would avoid many problems down the line.

Practically, in terms of workflow, I've seen a few different ways to make this effective:

- GitHub issues or any other collaborative document are great ways to post and discuss an initial design (using the SDL).
- Involve project managers, designers, and documentation specialists as early as possible in the process.

Once you've designed the schema for the new functionality, it's time to get some eyes on it.

Review

Schema reviews are a great way to ensure your proposed design makes sense to a variety of people. Hopefully, you have linters in place to help remove ambiguity and bike-shedding to keep your schema consistent. Instead, reviewers should think about the core design, something that automated tools can't do as well.

Who should the reviewers be? If you're reading this book, chances are you might be one of them! However, chances are that as your organization or team grows, the amount of "GraphQL experts" won't be enough for them to review all API changes going through every week and even every day. Review "teams" tend to work well at first, but it's hard to scale. Reviews taking more and more of your time are a big sign that it might be time to invest in schema/API quality tools to cut down on review time.

Overall our goal should always be to build quality tooling and great schema definition APIs that make it "hard to do the wrong thing" in the first place.

There is a point at which we have to trust that:

- Our team members have the proper documentation to learn about GraphQL and good API design
- The APIs and tools they use naturally guide them towards best practices.

Development

The development phase ideally begins once you have agreed on the ideal design for a new use case. If possible, try to avoid thinking of how to build it before that point, since we don't want implementation details to impact our design, even though that won't always be possible. There isn't much to add to the development phase. If you follow some of the advice we've seen along the book so far, you're on a good path!

Publish

You have confidence in your design, the team has implemented the GraphQL schema, and your tooling and reviewers have confirmed your schema is high quality. What next? It's time to start publishing it. If you're working with a single client, this step is usually quite straightforward and you can often go straight to deploying your change globally and letting the client integrate with your new use cases.

If you're dealing with a lot of clients, or even a public API, making changes like these might require more of a plan. The goal of this phase is to gain confidence in our design and implementation before opening the gates to all traffic. Let's cover three techniques, from more targeted to more public.

Mock Server

A mock server is a server that obeys to your GraphQL interface but does not actually serve or modify real data behind the scenes. It is incredibly useful to validate a design with known clients without having to fully deploy a change or risking making mistakes with real production data. Mock servers are not a new idea in the API world, but due to its type system, GraphQL makes it quite easy to build a "fake" interface.

The great graphql-tools utility by Apollo is a great choice to build a simple GraphQL mock server using JavaScript. GraphQL-Faker is another project that can be used as a CLI to quickly spin up a mock server given a GraphQL schema. It even lets us annotate the schema with custom directives to help the fake data generation:

```graphql
type Person {
  name: String @fake(type: firstName)
  gender: String @examples(values: ["male", "female"])
  pets: [Pet] @listLength(min: 1, max: 10)
}
```

You can push the idea further by integrating these tools or your own mock server tooling into your CI. If using a "schema as artifact" approach, our CI can pull in this artifact and provide teams a mock server URL before we have shipped the schema to production.

Feature Flags

Mock servers are great to share with internal teams, but not ideal to share with partners and external clients. We sometimes want to provide them with the real thing, without opening up to all our clients just yet. Perfect solutions to this are feature flags and schema visibility which we covered earlier in the book.

Feature flags let you work with select clients, and by using schema visibility techniques we explored, you can ensure these new use cases are not discoverable by existing and other clients. This is of huge help when validating a design. Working with an internal client at first is great, but an external eye often brings new concerns we didn't think of. The beauty of this approach is that since we only opened the new parts of the schema to select partners, breaking changes can be done by communicating with them directly. Something that is much harder to do when the schema was released to all clients already.

API Previews

Finally, API previews are similar to the feature flag approach, but intended to be used by anyone. API providers that support previews will often announce them publically and clients that are interested can onboard if they want, often using a special header. For example, GitHub uses Schema Previews to let integrators try out new features and hopefully give beneficial feedback before the schema graduates to general availability. They can be a really useful technique to gather feedback when you have a large pool of clients about which you don't know much, contrary to when working with select partners or internal applications.

The way they're often implemented is through HTTP headers. Clients can provide a `FooCorp-API-Preview: new-cool-feature` header which enables access to the API in preview. In GraphQL, this is often a part of our graph that is usually hidden. Once again, this is easy to do if you have schema visibility filters in place since it requires logic to hide or show parts of the schema at runtime:

```graphql
type Query {
  newFeature: NewShinyFeature!
    @preview(name: "new-shiny-feature")
}

type NewShinyFeature @preview(name: "new-shiny-feature") {
  field: String!
}
```

The implementation is almost exactly the same as a feature flag, but the idea behind them is different. I must warn you that while API previews can be a good way to gather feedback for a new use case, they can be quite easily overused. Make sure previews are graduated quickly or else they often naturally become part of the API in a weird state where a ton of clients use the field, which means we can hardly change it anymore, but the field is still tagged as in "preview mode". Make sure you know what you want to get out of a preview period, that you gather information from users and data, and that you graduate them to GA as soon as possible once the idea is validated.

Previews can be useful for public APIs, but for an internal or partner APIs, feature flags are generally a much better idea.

Analyze

We've designed a great schema, implemented it, and published it to limited audiences and/or the world! We're not done yet. If you have implemented some of the schema analytics we talked about in the tooling chapter, you can observe how these new use cases get used over time. How's the performance of the new functionality? Is it getting used? Can we reach out to the biggest users and see what could be improved? These are all great questions you can answer really well, thanks to GraphQL's declarative nature. Often, new use cases and functionality can be found from looking into workarounds that integrators have to go through to achieve their goals. Do reach out to teams and clients using your API, it's time to make crucial changes before our new functionality becomes available to everyone.

Ship

It's time to ship your new API to everyone. If you've designed carefully by thinking of actual use cases and bringing in stakeholders and domain experts, ensured quality with code review and automatic linters, published to limited audiences to validate functionality and minimize risk, and finally, monitored and analyzed the usage of your new feature, you're ready to go! Keep analyzing performance and usage. You will learn more as more and more clients integrate with your API.

Public GraphQL APIs

We can often categorize APIs in three broad categories:

- Private APIs (Internal to an organization)
- Partner APIs (Shared with a limited amount of partners)
- Public APIs (Fully public APIs accessible by almost anyone)

It has been interesting to see that GraphQL is mainly very popular for private and partner APIs, and a bit less in public APIs. Even Facebook doesn't have a public GraphQL API at the moment of writing this book, and a lot more people are using it internally. As far as I know, Shopify and GitHub are probably the largest public GraphQL APIs currently. Why are public GraphQL APIs so rare? I think there are a few reasons.

Is GraphQL a Good Choice for Public APIs

> The target audience should be the single biggest influence on your API Design.
> *Daniel Jacobson, when talking about Netflix's approach to APIs*

We can certainly ask the question of whether GraphQL is a good idea or not when implementing an open or public API. First, looking at where it comes from, we know that Facebook still has no public GraphQL API and that GraphQL was architected of other problems than the ones that come from GraphQL. An example I always bring up, when talking about how GraphQL was initially more oriented towards internal APIs, is Facebook's GraphQL client Relay. Since Relay made many assumptions about the design of a GraphQL server, our APIs had to be designed in a certain way if we wanted to support clients that were using Relay. As we saw in the chapter on design, many of these things are generally good ideas (input and payload types, connections), but the fact that a client was so coupled to a specific server implementation means that an API provider must take a specific type of client into consideration.

As a public API provider, we can't assume that all our clients will be using Relay, which means a lot of clients are forced into fields that are Relay specific. I remember even thinking about building a Relay specific version of a schema at some point or a special relay field on the root. Quite a hurdle for public API providers! Since then, the new "Relay Modern" implementation has made a lot of these assumptions best practices rather than hard requirements. This lets clients use Relay with servers that did not necessarily choose to implement every Relay recommendation. A lot of these recommendations have also since moved to be GraphQL best practices in general.

However, that doesn't mean GraphQL can't be a great alternative for public APIs as well. GraphQL can be a really powerful way of exposing possibilities to a large set of clients. A public API (if successful) often means a ton of possible clients. Handling so many various use cases can become quite hard for API providers, and failure to handle these use cases becomes cumbersome for clients, leading to very generic and one-size-fits-all APIs. GraphQL is fantastic at handling such situations. Together, the type system, resolver pattern and query language allow providers to define as many capabilities and variations as they want without extra cost on existing clients (for example, think about typical enormous webhook or HTTP endpoint payloads). GraphQL also offers a good and most of the time simple mental model and architecture for API engineers.

The issue is that public APIs are often more generic by nature because we don't always know nor do we want to predict all the client use cases. Public APIs are usually generic in a way that lets clients the possibility of inventing new use cases through very generic interfaces. Of course, that often leads to the One-Size-Fits-All problems, so a public GraphQL API has to try and strike a balance between offering functionality that is generic enough for clients to build solutions on top of our API platform, but also avoids falling into the same "One-Size-Fits-All" trap many have criticized REST for. After all, even Roy Fielding, author of the REST dissertation, acknowledges that trade-off:

> The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.

GraphQL has the potential of being finer-grained and more client-specific, but this means we have to design the schema in this way in the first place. It would be impossible to create a field per client when dealing with hundreds of thousands of clients, so the answer must lie somewhere in between. As covered in Chapter 2, aiming for more specific fields is often a good idea. Very generic fields, full of bells and whistles, should probably be avoided. The key is often listening to clients and not hesitating to express new use cases as new fields rather than overloading existing ones.

Lack of Conventions

Conventions often make great public APIs and it turns out HTTP is quite a good standard. When using GraphQL as our public API, we often give away a lot of what APIs that lean more on the HTTP specification often give us.

A lot has to be reinvented with a GraphQL API. Errors, caching, rate limiting, timeouts, and so many other things are handled in many different ways by GraphQL API providers at the moment. Part of it is because best practices are still emerging, but the other part is because the GraphQL specification doesn't say anything about it, which is wise in many ways. Currently, a GraphQL client must learn how a certain provider handles all these elements. Is the GraphQL server using errors in data, or using GraphQL errors? Is the GraphQL API returning 429 or does it have a custom error encoded when a client gets rate limited? What about cache hints? Every time a client interacts with a new GraphQL API, they have to find how all these things must be handled, either through exploration or documentation. Compare this to an HTTP API, where you can most of the time expect the right behavior when you throw an HTTP client at it.

Of course, a lot of this is a challenge to typical HTTP APIs too, which is why you'll always hear the advice of using as many HTTP concepts as possible before implementing your own logic. Should public GraphQL APIs served over HTTP return a `429 TOO MANY REQUESTS` or more something like this?

```
{
  "errors": [{
    "message": "Rate Limited",
    "code": "RATE_LIMITED",
  }]
}
```

With 429, most of HTTP libraries will handle these requests perfectly. With the other approach, we require clients to learn a whole new "protocol" for our API, try and learn about every error possible and how it's expressed in this particular API. This also explains why generated clients are so popular with things like gRPC, since companies can generate their own concepts into clients instead of letting users with generic clients try to handle these scenarios.

My advice would be for GraphQL APIs to rely on conventions as much as possible. Clients should not have to learn new conventions every time they face a new GraphQL API, especially if we want GraphQL to succeed as a technology.

With Great Power comes Great Responsibility

There is something else to keep in mind: we love the fact that GraphQL puts the power in the client's hands. However, this is also a source of concern: with this power, responsibility should be in the client's hands as well, but that's rarely the case when it comes to public APIs. Take monitoring, for example. With GraphQL, the easiest way to find regressions and monitor performance would probably be on the client side. However, as a public API provider, we don't want to wait for client support requests coming in before acting upon performance regressions. As we covered in Chapter 5, this leads to very complex monitoring solutions trying to understand how every query possibility is behaving. That is way easier to handle with a smaller set of known, internal queries.

GraphQL works beautifully when you control the clients since so much logic is moved at this layer. With a public API, we lose a lot of that control, which greatly increases the complexity. It is easier when you know those clients, or even better if you can talk to them because they are across the room. When they are hundreds of thousands and you don't have great communication channels, things can get ugly.

Overall, GraphQL's sweet spot is currently still around supporting a set of known clients with diverging needs. It can offer amazing benefits in a public API setting as well, but know this is far from a solved story at the moment. I'm hoping we see more conventions and tooling that make it easier to support open API using GraphQL.

Summary

In summary, as shown by both GitHub and Shopify, GraphQL **can** be a good choice for a public API. However, this choice means that we need to be careful when designing our schema to not fall in the very tempting One-Size-Fits-All trap. Finding the right balance between generic and specific is key, but don't be afraid of providing multiple ways of achieving use cases since it does not affect existing clients.

Finally, conventions are still far from being set in stone. As more and more providers turn to GraphQL for public APIs, it will be important to make sure clients can expect consistency from a GraphQL API.

GraphQL in a Distributed Architecture

In this chapter, we will be taking a look at different popular distributed architectures and strategies around GraphQL. Over the past few years, while its origin comes from a monolithic API layer, GraphQL has been used in many different contexts such as service-oriented architectures.

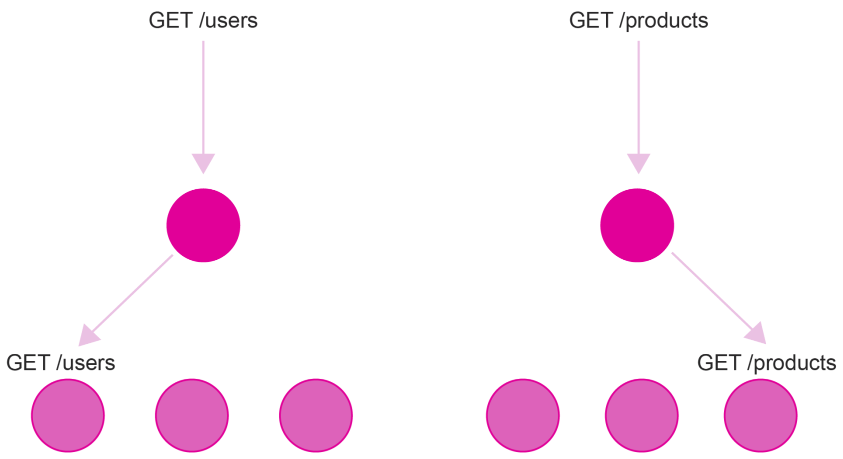
GraphQL API Gateway

GraphQL was very quickly applied to the API gateway pattern. An API Gateway is usually a service that acts as an entry point to experiences/functionality. It is often seen in a distributed architecture to decouple use cases that clients are interested in and are consuming from the underlying services involved in these use cases. Since the entry point should aim at hiding the implementation details of use cases, clients don't need to know how many "services" are behind a given functionality. The gateway allows a provider to change these details over time while maintaining a stable facade.

Providers can also unify a lot of logic like rate limiting and authentication, instead of implementing this same logic on all API servers. Some more complex API gateways are also able to aggregate what would have been multiple calls to individual services into one request. In that sense, it definitely seems like GraphQL can be a great abstraction over existing APIs, providing a schema to expose these possibilities, and the query language to allow clients to consume the use cases as they want from multiple sources, behind the scenes.

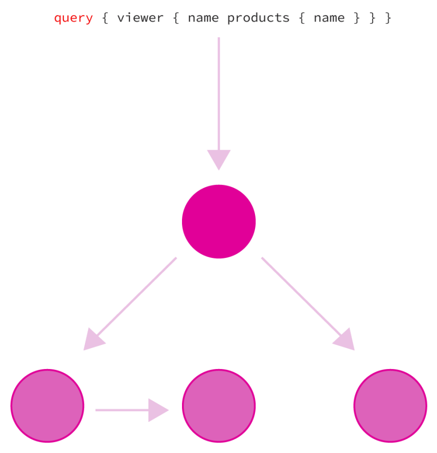
There are a few concerns with API gateways that we should keep in mind while analyzing solutions. The first one is that it is often very tempting to do too much at the gateway level. Because of that, API gateways often become a centralized point of failure, and centralized configuration, dealing with concerns that individual services should be in control of. After all, the reason most teams turn to micro-services is to decentralize this logic in the first place. In fact, in an ideal world, our API gateway would probably be closer to just another service, rather than this big centralized point in our system (easier said than done).

What makes GraphQL API gateways particular is that they're usually not as simple as when dealing with different typical HTTP resources. A lot of API gateways that deal with endpoint-based APIs act like a simple proxy, meaning that it's simple to map a request to an underlying service.



In the above example, the API gateway resolves the `/users` endpoint by calling down the appropriate service, and the `/products` resource by calling down to another service. Note that while we compare it to a proxy in terms of implementation, conceptually, an API gateway should not just be a proxy since that would mean our gateway is very coupled to the underlying implementations, we should rather aim for a "door" to use cases rather than just a "dumb proxy".

With GraphQL, a query often can't be resolved through one unique underlying service and chances are fields are querying across multiple services. You can imagine that when queries get larger and more complex, the GraphQL gateway has to orchestrate a complex plan to fetch this information:



It's important to say that the simple case is more common with endpoint-based APIs. Some of the same problems arise when gateways start composing different requests together or making transformations to the underlying service representations for example. GraphQL simply shows that problem even better. There is still a lot of value in providing a GraphQL interface over our domain, so how do we deal with these challenges? In the next few sections, we'll explore the most common solutions when building a GraphQL gateway. As we cover different approaches, it is good to keep in mind what a great API gateway looks like, and common pitfalls to avoid. Now let's cover some implementations.

The "Simple" Way

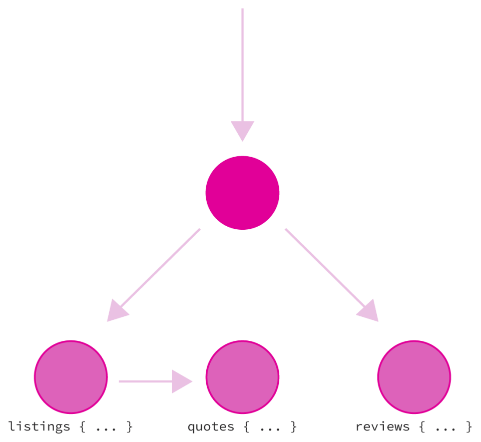
The simple "proxy" resolution for a use case is appealing due to its simplicity, and while GraphQL queries often tend to span multiple areas of responsibilities and services, that's not necessarily a hard rule and the way we design our schema can make the execution of queries simple. In fact, the first time I saw an organization use GraphQL as a gateway was probably from a blog post by Airbnb. Airbnb had existing Thrift schema for their use cases, separated across different presentation services. To expose these use cases as a united GraphQL interface, they translated Thrift interfaces to GraphQL schemas, and simply merged them together in a way that makes execution more straightforward for a gateway. Take a look at a query from the blog post:

```graphql
query LuxuryHomeQuery {
  luxuryHome {
    listings: luxuryListingsById(listingId: 123) {
      id
      bathrooms
      bedrooms
    }

    reviews: reviewsByListingId(listingId: 123) {
      // ...
    }

    quote: luxuryListingQuote(listingId: 123) {
      // ...
    }
  }
}
```

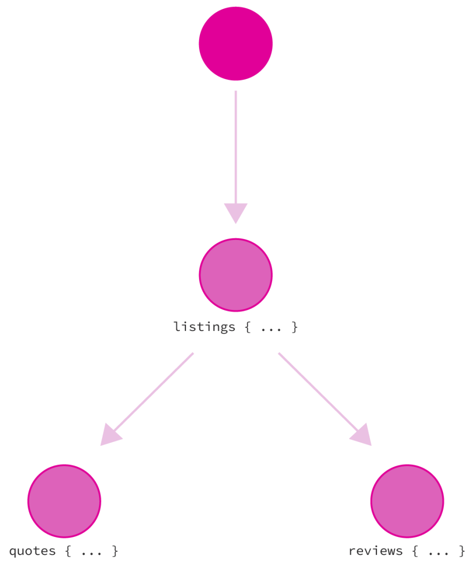
One thing you notice here is that we are fetching three distinct concepts related to a single `listingId` in this query, `listings`, `reviews`, and `quote`, but that we have to provide the `listingId: 123` every time we query something related to it. Compare that to how a typical GraphQL server would choose to design this use case:

```graphql
query LuxuryHomeQuery {
  luxuryListings(id: 123) {
    id
    bathrooms
    bedrooms
    reviews {
      // ...
    }
    quote {
      // ...
    }
  }
}
```

GraphQL is great at allowing to fetch relationships from a certain node. Why has Airbnb chosen to design the schema in a way that almost resembles typical, more RPC, endpoints? They don't talk much about the reasons why in their post, but we can see that in the first case, the execution plan for a gateway is **much simpler** than in the second. With Airbnb's approach, we can imagine the execution looks quite similar to the simple proxy approach, being able to separate parts of the query easily between underlying services:



If they chose to design the schema in a more “pure GraphQL” way, they would have to handle a much more complex execution plan for the query. The service in charge of fetching a listing would then have to understand the rest of the query to fetch the reviews and quote, or resolve the listing field, return it back to the gateway, for the gateway to then query the missing information. In both cases, that’s a lot more to handle:



The downside to this approach is that we definitely lose some of the beauty of the declarative GraphQL query language and go back to what looks more like "batch requests". The beauty of it is that our API gateway can remain simple without doing too much or being too coupled to the downstream services. The gateway simply "wired-up" the different parts of the schema together.

Schema Stitching

Airbnb's approach of wiring up different schemas together to form the API gateway is commonly referred to as "schema stitching", the act of taking two or more parts of a GraphQL schema and combining them to make a valid, unique GraphQL schema. For example, two services could define their part of the schema, the one they own:

```graphql
# User Service Schema

type Query {
  user(id: ID!): User
}

type User {
  name: String!
  age: Int!
}
```

```
# Product Service Schema

type Query {
  product(id: ID!): Product
}

type Product {
  name: String!
  price: Int!
}
```

Schema stitching would take these two schemas and merge them as one, like this:

```
type Query {
  user(id: ID!): User
  product(id: ID!): Product
}

type User {
  name: String!
  age: Int!
}

type Product {
  name: String!
  price: Int!
}
```

As you can see, types were merged under the same schema, and shared types (like `Query` in our example) merged fields. This example is very similar to the Airbnb example we covered right before, as it is very easy for a gateway to look at a query and see which service should resolve it. There are no **type dependencies** between services. Now imagine we want to query the products a user has for sale. We would need to modify our schema so it looks more like this:

```
# User Service Schema

type Query {
  user(id: ID!): User
}

type User {
  name: String!
  age: Int!
  productsForSale: [Product]!
}
```

Quickly, we realize we have an issue. The schema from the user service is invalid since it uses the `Product` type, which is actually on the product service schema. It only becomes valid when the schemas are stitched together. To solve these type dependency issues, schema stitching is usually done at the gateway level, combining schemas but also adding **links between schemas**. Using Apollo schema stitching, the logic looks like this:

```
const linkTypeDefs = `
  extend type User {
    productsForSale: [Product]!
  }

  extend type Product {
    owner: User!
  }
`;

mergeSchemas({
  schemas: [
    productSchema,
    userSchema,
    linkTypeDefs,
  ],
});
```

We're not done, we've now extended the `User` type with a `productsForSale` field, but how is it resolved? We need to write some resolving logic that will take our user object and query our product service for the rest of the query. Again, using Apollo's solution, it looks a bit like this:

```
const mergedSchema = mergeSchemas({
  schemas: [
    userSchema,
    productSchema,
    linkTypeDefs,
  ],
  resolvers: {
    User: {
      productsForSale: {
        fragment: `... on User { id }`,
        resolve(user, args, context, info) {
          return info.mergeInfo.delegateToSchema({
            schema: productSchema,
            operation: 'query',
            fieldName: 'productsByUserID',
            args: {
              ownerId: user.id,
            },
            context,
            info,
          });
        },
      },
    },
    Product: {
      owner: {
        fragment: `... on Product { ownerId }`,
        resolve(product, args, context, info) {
          return info.mergeInfo.delegateToSchema({
            schema: authorSchema,
            operation: 'query',
            fieldName: 'user',
            args: {
              id: product.ownerId,
            },
            context,
            info,
          });
        },
      },
    },
  },
});
```

There are a number of issues with this approach that eventually led Apollo to mark schema-stitching as **deprecated**:

- We talked about how a gateway should know as little as possible about the underlying services. The gateway should be a thin layer. In this case, we have to define relationships and even handle how they're resolved. We wanted to decentralize these things, but now all the logic for cross-service queries is centralized at the gateway.
- The gateway logic can be quite brittle and needs to handle things like name conflicts and to make sure the underlying services don't change their schemas.

Apollo's Schema Federation

Schema Federation is a product Apollo released that aims to replace schema stitching. We won't go into detail on how it works, but the idea is to avoid the pitfalls of the schema stitching approach. Federation aims for full decentralization of the schema by letting individual services define their own schema and the way they **extend** the others, removing the need for gateway defined links and business logic. To make this work, federation is a specification that lets us annotate type in a way that lets the federating gateway resolve queries across services by building a "query plan".

There are several things to look out for when opting for an approach like schema federation. As we covered in Chapter 5, predictable performance is always a challenge, even in a monolithic GraphQL server. Now we're dealing with query plans, something that is infamously hard to control and tune. It can become a black box that is very hard to debug. It's up to the implementation and tooling to make that easier. The other thing is that all your services need to implement the federation spec for it to work.

If I were to use schema federation in a project, I'd make sure to be careful about a few things:

1. Your underlying services may not be the right split for your federated graph. Use cases sometimes span multiple services, and you should probably aim to divide your graph by use cases and sub-domains rather than how your services are split up. If you have 100 services it doesn't necessarily mean you want 100 separate schemas.
2. Keep the "links" between schemas as simple as possible to avoid unpredictable performance.
3. The most mature (and original) implementation of the specification is in JavaScript. At the moment, Federation is most effective if you're willing to build or already have JavaScript services. More and more languages are getting added, so keep an eye out.

With that in mind, federation can be a beautiful solution if it fits your team's needs and scales better than schema-stitching. The Apollo team has enormous

GraphQL knowledge and experience and will most likely keep improving the query planning and making it less of a black box.

Single Schema Gateway

Overall, most issues with stitching and federation stem from the fact that GraphQL is inherently a centralized approach to APIs. We can try to decentralize as much as we want, but the reality is that we still end up with centralization, either through a very complex gateway or by building a single GraphQL server. I truly believe GraphQL can have a place in a micro-services context as a single schema and thin API server.

What if we considered our GraphQL API as "just another service", something we wanted from a gateway in the first place? If you're looking to keep things boring and simple, one can build a GraphQL API server that is **resolved** by many underlying services, while still keeping the graph in a single place.

We're trading organizational complexity with operational complexity when we opt for something like federation or stitching. Both approaches can work. Facebook, GitHub, and Shopify all operate monolithic schemas and approach organization issues by building great developer tooling, rather than by separating the graph in different repositories.

My preferred approach for building a GraphQL gateway today would probably be a thin GraphQL facade operating as a single service. A GraphQL API can act as a gateway without necessarily using more complex approaches like stitching or federation. We can keep the schema and interface centralized, but federate the execution (the resolvers). We must be very careful of leaving business logic almost entirely out of the gateway itself. I really like this approach because I think it's easy to adopt it progressively for teams with **existing** architectures. A lot of organizations already have service communication in place using other approaches like REST or gRPC. It's easy to reuse these pieces when building our GraphQL this way, rather than having to implement GraphQL APIs on all services like stitching or federation currently requires.

Unlike stitching and federation however, there isn't really any off-the-shelve mature GraphQL gateway implementation like the one I'm describing here. By following most of what we've covered in the book so far, we already have most of what is needed. With something like GraphQL, keeping things boring and simple is actually a very good thing!

One of the ideas behind service-oriented architecture and schema federation is to decouple the teams when it comes to developing software. Federating the schema means different teams can evolve the schema on their own side of things, but allowing the gateway to merge all of these under a single API down the line.

While it may help teams make changes to the schema faster due to smaller code-bases, we must remind ourselves that we are still interacting with a centralized graph in the end. These problems must be addressed at the core, by making sure teams are naming things properly and not reusing generic types across the board. Naming conflicts are often brought up when trying to merge schemas together. This is usually a symptom of inapropriate naming in the first place.

Separation of concerns and namespacing are things that can be addressed to best practices and tooling whether we're building our schema in a single codebase or if we federate it across many. My recommendation is to use linting and best practices to help teams name their concepts appropriately, and the tools your programming language offers to decouple parts of the schema.

GraphQL as a BFF

We've talked about the "Backend for Frontend" pattern in the introduction of Chapter 1, and how GraphQL aims to address some of the same problems. We even compared GraphQL as some sort of BFF. Let's dive deeper into this. What exactly is a BFF? A BFF really can be considered as some sort of API gateway.

At its core, the BFF pattern is an answer from SoundCloud to common problems that may appear when building OSFA (One-Size-Fits-All) APIs, where we have a single API that aims to answer the use cases of many different client applications. In an OSFA API, lots of different client applications often have to share server-side resources, which can often lead to issues evolving in? from? use cases on the backend, and poor experience from the client perspective.

In the BFF pattern, we choose to acknowledge there are differences between clients and to build individual API servers **per client or experience**. These servers are then free to be more specific to a client's need, and free to evolve without the issues we face when trying to share resources. But the BFF pattern is not just a technological choice, but often an organizational one as well. Teams that are in charge of a particular "experience", for example the mobile team, has full control over that API and the client, which avoids some of the organizational problems and communication issues that often arise with OSFA APIs. Note that these problems are solvable in different ways, but the BFF pattern simply takes this opinionated stance on solving the issue.

Reading this, we can draw some parallels with the benefits of a GraphQL API. With GraphQL, adding fields has no impact on existing clients, which does make evolution a bit more approachable and which enables one server to support many different clients. However, besides the fact clients need to declaratively select their needs, there is nothing that makes GraphQL inherently a good replacement or similar solution as a BFF. In fact, the way we build our GraphQL server could just as well turn into an OSFA GraphQL API.

The other thing to note is that BFF goes further than just optimizing for client use cases on the representation side of things. A BFF may decide to serialize payload differently, caching things differently, authenticate in another way, etc. To really have a "GraphQL BFF" pattern, we would most likely need multiple GraphQL servers, in which case we would be really just implementing BFF in its original form.

Still, a GraphQL server can definitely be used to optimize multiple client needs within a single server, which can be incredibly useful. There are certain aspects to keep in mind when doing this:

- A GraphQL "BFF" needs not to fear adding multiple distinct ways of achieving similar things. This could be adding client-specific fields, and even a completely different type if needed. The beauty of this is that existing clients are not affected at all. The gotcha is that our schema does get some more "bloat" and documentation, and discovery might be affected.
- We must avoid the temptation of building a generic OSFA GraphQL API.
- Independence must be achieved through tooling and culture since the different experiences are not as naturally separated as with the BFF pattern.

Another thing to note is that common issues with the BFF pattern are shared concerns. That's one benefit of maintaining a single GraphQL server when it comes to being able to optimize for multiple experiences. We can still reuse the same authentication system, rate limiting logic, authorization system, etc. As we covered earlier though, this could be seen as a downside just as well as an advantage, because we may also want these concerns to be experience-specific. This is highly context-dependent.

All in all, GraphQL can definitely be used as a single BFF, and employed to support multiple client experiences when it comes to the representation of resources. However, the main goal of the BFF pattern, full autonomy of those experiences, has to be maintained more actively than with multiple BFF servers, which are by nature more isolated and independent.

Service Communication

As we already covered during this chapter, GraphQL can be a great choice in terms of serving an API to North-South traffic, as an interface to our system. But is it a good idea to use GraphQL to communicate between services, in an "East-West" manner? First, let's look at the requirement of good service-to-service communication.

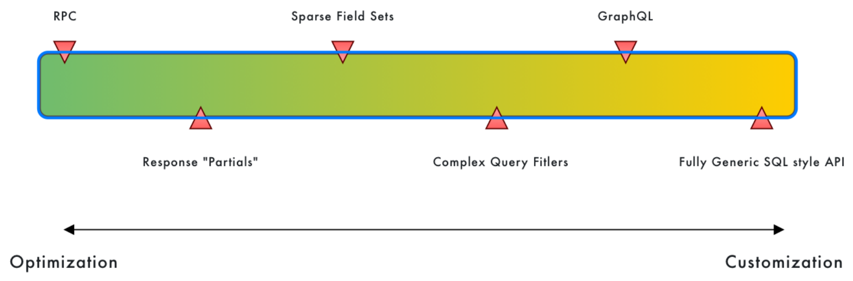
Network boundaries are quite useful for a number of reasons, namely independent scaling, availability, and development speed. All these are good reasons for which some of you have opted for a service-oriented architecture. However, when we go from extremely fast method calls to network calls, we introduce a lot of overhead. This is why one of the first goals for someone thinking about service-to-service communication should be to reduce this overhead as much as possible. For this reason, performance is quite important when choosing a protocol/architecture for communications.

Another thing we have to be careful about is to realize that remote calls (over the network) are inherently different than local calls (method calls). They usually require a very different design. Fortunately, pretty much any technology choice available these days will allow us to design our API in a way that makes remote calls possible, as long as we're careful with our API design. This also means we need to think about resiliency and to refrain from abstracting the network too much, in order to avoid making network calls behind the scenes or making things inefficient without realizing it.

API evolution is also a big issue. We don't want a field that was added to one service's API requiring changes to all clients in our system. Expand-only types are very useful. They help us avoid lock-step deploys. Most API solutions today offer these qualities, GraphQL being one of them. But that does not point us towards a specific solution just yet.

Another thing we may want to take a look at is the **protocol** we want to use. Most people give GraphQL a hard time since most APIs are implemented over HTTP 1.1. While that's probably true, it doesn't have to be this way. GraphQL is purposefully transport/protocol agnostic. Nothing is stopping us from implementing GraphQL over H2, over UDP, or even as a new GraphQL specific protocol. However, the counter argument is that it hasn't been really tried before, and chances are some semantics we're used to (200 Status Code, headers) with current GraphQL APIs would need to change or at least be adapted. Generally, I like to think that protocol or transport layer is mostly a non-issue if we had strong enough incentives to use GraphQL in Service-To-Service communications.

To see if GraphQL would be a great fit or not, maybe we can take a look at what it's great at and what the tradeoffs are. One way I like representing these tradeoffs is with an API customization spectrum:



Different API styles often have to pick between how flexible they are and how optimized they are. Some styles are very far on one end. For example, RPC is usually a very optimized function call, and GraphQL is on the other extreme, allowing for a lot of flexibility on the client side.

On the optimized side, we've got a lot of advantages:

- Caching is more effective.

- Server-side performance is usually better (since we only have one thing to optimize).
- Very clear use cases.

But also disadvantages:

- The One-Size-Fits-All problem we talked about in this chapter
- Bandwidth and latency may become an issue with large, generic payloads

On the flexible or customizable side, we've got the usual GraphQL benefits we've been talking about so much already:

- Bandwidth savings.
- Easier to support a larger amount of client use cases.
- Adding to resources comes at less of a cost.

But that also comes with a set of drawbacks:

- More complex execution and performance for the backend.
- Not as clear and focused use cases.

All in all, I don't think service communication is the sweet spot for GraphQL usage. Technologies with RPC in mind like gRPC were made for these usecases

Summary

GraphQL is a beautiful abstraction for APIs and is tempting to use in a distributed architecture. Keep in mind that GraphQL execution is not trivial, and executing a query in a distributed context is likely to make things even more complex.

If you operate a service-oriented architecture and are looking to expose a GraphQL API here's my recommendation:

1. Consider building a GraphQL API service, as a single schema, that communicates with underlying services to resolve the use cases. Keep it as thin of a layer as possible and make developer experience great through tooling and best practices.

2. Even though Apollo's stitching approach is officialy deprecated, I believe it's still a valid approach under one condition: that you only need to merge fields at the `Query` root. Any nested links between schemas is really annoying to maintain, as we saw earlier in the chapter.

3. For anything more complex than merging at the query root, Apollo Federation is currently the best approach for a gateway with an inteligent query planner.

Finally, I believe there are more effective communication patterns for service-to-service calls like gRPC, and the benefits that GraphQL gives us don't apply as much in that context.

Versioning

Versioning is probably one of the topics that triggers the most questions about APIs and the one that brings the most controversy and heated discussions.

How do you version GraphQL APIs? The most common answer you'll get these days is "you don't". If you're like me when I first read that, you might be a little anxious about maintaining a version-less API or a bit skeptical of the approach. In fact, not only is this a common answer, but it's listed as a main feature of GraphQL graphql.org's landing page..

When reading graphql.org, this may look like a feature being specific to GraphQL, but in fact, evolving APIs without versioning is something that has been frequently recommended for REST and HTTP APIs already. In fact, Roy Fielding himself famously said at a conference that the best practice for versioning REST APIs was not to do it!In this chapter, we'll explore API versioning in general and try to see which evolution approach makes the most sense for GraphQL APIs.

API Versioning is Never Fun

A good way to understand where this idea of not versioning an API at all comes from is to look at how web APIs are typically versioned. The truth is that in any approach we end up choosing, we accept the tradeoffs they come with. Probably the most popular approach out there is to globally version APIs. Think `v1`, `v2`, `v3` every time we have to make changes that could potentially be breaking to our integrators. The approach makes sure existing customers never get broken by some of these changes, but comes with serious annoyances:

- Every time we release a new major version, we leave every single client on older versions behind, forcing them to upgrade for those newer changes.
- Most providers will avoid making too many major version changes. So, when new versions are deployed, they often come loaded with a ton of changes, making it harder for clients to understand and to basically re-integrate with the API from scratch.
- The API provider needs to choose between supporting older API versions forever, leading to increased complexity, or to start breaking clients that are stuck to older versions.

In practice, this global versioning is done in different ways:

- In the URL: `v1/user`, `v2/user`: Global versioning using the URL often leads to an explosion of new resources (even though most of them remain unchanged) under the new URL version. It leads to a massive amount of URLs to support over time. This doesn't play well with caching and clients often need to rewrite everything, since they can't trust that their existing `v1` resources will play well with the `v2`.

- Using headers: `Stripe-Version: 2019-05-25`. Header-based approaches often try to make smaller changes between versions, but the overall problems still remain. Providers need to find a way to support multiple versions while keeping the complexity as low as possible. A great example of this is Stripe's approach to versioning.

Overall, global versioning might make a lot of sense when we are in fact dealing with a new set of resources or a completely different API.

An alternative that is not as "global" as these two approaches is to apply versioning content negotiation. For example, the GitHub API allows clients to specify a custom media type that includes a version number. This is a pretty good way to express versioning for the representation of a resource, but not necessarily for other types of changes. The other downside is that clients aren't forced to pass that media type, and if it goes away, they are usually reverted back to the latest version, almost certainly breaking their integrations. For GraphQL, since we aren't dealing with different HTTP resources, this is a no go anyway.

Versioning GraphQL is Possible

Even though the "no version" approach is often recommended, nothing is stopping anyone from versioning a GraphQL API. In fact, Shopify adopted a URL versioning approach to evolve their GraphQL API.

Their versioning approach uses the URL for global versioning. With GraphQL, this is less annoying because it doesn't create a full new hierarchy of resources under the new identifiers. It uses finer-grained, 3-month long versions, which helps with the typical global versioning problems.

Practically, making it work is quite challenging. As discussed in Chapter 3, maintaining multiple schemas can be challenging. One way to make versioning work would be to build completely different schemas depending on the URL or version a client is requesting. You can imagine this would cause a lot of overhead to server developers and becomes incredibly hard to manage as the number of versions piles up. Instead, a runtime visibility approach as we discussed earlier in the book may be used to present users with different variations of the schema. This comes with a lot of complexity as well but is possible.

The same kind of catch 22s we saw with global versioning applies to GraphQL as well. The fact that all these approaches lead to broken clients down the line or incredible complexity on the provider side leads to an often recommended approach: simply aiming for backward compatibility at all costs, opting for additive changes when possible, and thinking of extensibility instead of breaking changes.

If you're like me when I first read about this approach you're probably thinking "but sometimes changes are unavoidable!" It's true, even with all the best practices kept in mind, mistakes can happen. But as we'll see, whether we version an API or not, mistakes happen anyway.

While versioning often gives a sense of security to both providers and clients, it doesn't generally last forever. Unless an infinite amount of versions is supported, which causes unbounded complexity on the server-side, clients eventually need to evolve. In that Shopify example, this happens 9 months after a stable version is released. After those 9 months, clients need to either upgrade to the next viable version, which contains a smaller set of changes or upgrade to the newest version, which probably includes a lot more changes.

Continuous Evolution

The alternative to versioning, as mentioned at the beginning of this post, is to simply not do it. The process of maintaining a single version and constantly evolving it in place, rather than cutting new versions, is often called Continuous Evolution. One of my favorite ways to describe the philosophy comes from Phil Sturgeon:

> API evolution is the concept of striving to maintain the "I" in API, the request/response body, query parameters, general functionality, etc., only breaking them when you absolutely, absolutely, have to. It's the idea that API developers bending over backwards to maintain a contract, no matter how annoying that might be, is often more financially and logistically viable than dumping the workload onto a wide array of clients.

A big part of the philosophy behind continuous evolution is a strong **commitment to contracts**, aiming to evolve the API in a backward-compatible way the absolute best we can. This may sound like an unachievable dream to some of you, but I've noticed a lot of API providers will settle into versioning rather than trying to find creative ways to maintain their interface.

Additive changes are almost always backward compatible, and a lot of the time breaking changes can be avoided if they are used wisely. The main downsides to an additive approach to evolution are usually naming real estate and API "bloat". The naming issue can usually be mitigated by being overly specific in naming in the first place. The bloating, especially in GraphQL, is most probably less of an issue than the cost of versioning to clients. But that's of course a tradeoff for API providers to decide.

But as we covered earlier, some changes are simply unavoidable. Not all changes can be made through addition, and there will always be moments where a breaking change needs to be made. In fact, here are a few good examples of unavoidable breaking changes:

- Security Changes: You realize that a field has been leaking private data or that a certain set of fields should never have been exposed.
- Performance issues linked to the API design: An unpaginated list that can potentially return millions of records, causing timeouts and breaking

clients.

- Authentication changes: An API provider deciding to deprecate "basic auth" altogether, forcing API clients to move towards JWTs.
- A non-null field actually can be null at runtime. This causes errors that are not fixable without breaking the schema contract.

In these four example cases, there is often no way an additive change can be made to address the situation. The API must be modified, or fields be removed. With continuous evolution, we rely on deprecation notices, a period to let clients move away from the deprecated fields, and a final sunset making the breaking change.

Versioning seems like it would solve breaking change issues but if you look at the examples we listed, none of them would be easy even if we had a great versioning strategy in place. In fact, we would need to make breaking changes in all affected versions, making use of deprecations, a period to let clients move away, before finally making the breaking change. Notice something funny? Versioning requires the same amount of work (possibly more, depending on the number of concurrent versions) than if we had a single continuously evolving version.

With all these strategies, it's how we go through that deprecation period that will determine how good our API evolution is. This is sometimes referred to as "change management".

Change Management

As API maintainers, no matter what evolution/versioning process we decide to standardize, one thing is certain: we have to get good at change management. GraphQL gives us a few really great tools to become skillful at this. The first step to a good change management strategy is to make the upcoming change visible to existing integrators.

Deprecations

Deprecations are the best tool for continuous evolution. They allow us to mark certain parts of our API to say that their usage is not encouraged anymore, and potentially going away in the future. GraphQL comes with a great tool for that: the `@deprecated` directive:

```graphql
type User {
  name: String! @deprecated(
    reason: "Field name is being replaced by field `username`"
  )
  username: String!
}
```

The `@deprecated` directive is incredibly useful when it comes to moving away from fields that will become unavailable in the future. It has a `reason` argument to allow providers to explain the reason behind this upcoming change. Unfortunately, at the moment, this directive is only applicable to fields and enum values, but will soon be coming to arguments and input values.

The great benefit of `@deprecated` is that it is a respected standard across clients, which means tools can help us. For example, GraphQL will hide any deprecated fields by default, to avoid having new clients onboarded to a field that is going away. GraphQL clients can start throwing warnings when using a deprecated schema member (member or members?), and documentation generators can automatically annotate these members as well.

This is great, but we can still make it better. In my experience, a great deprecation requires additional details:

- The date at which the schema member is going away (sunset date).
- The reason why the schema member is going away.
- An alternative to that schema member.
- Optional: a link to a longer form explanation (A blog post for example).

If you're using a code-first approach, we can encode all that information within a simple custom helper:

```
new GraphQLObjectType({
  name: 'User',
  fields: {
    name: {
      type: GraphQLString,
      deprecationReason: deprecationReason(
        reason: "Name is going away",
        alternative: "Use `username` instead",
        sunset_date: "2030-05-01",
        link: "https://dev.gql.com/blog/user-name-deprecation",
      )
    }
  }
})
```

`deprecationReason` here is a custom function we've created to ensure consistency with our deprecation messages and to make sure our developers always provide the necessary information when deprecating a schema member. In this particular example, the deprecation reason could end up looking like this:

```
Name is going away. Use `username` instead. Sunset date: 2030-05-01
For more information: https://dev.gql.com/blog/user-name-deprecation
```

This is a strategy we used at GitHub. When hundreds of developers are working on a GraphQL schema, utilities like these ones ensure a high quality and consistent experience for integrators.

Communication

Deprecations help us communicate changes, but they're often not enough to make all clients aware of upcoming changes. The best API platforms are able to communicate changes through many different means, including deprecations. This is where the approach of tracking all queries comes in very useful.

One of the biggest advantages to GraphQL is the fact a provider knows, down to every single piece of the query, how their API is being used. If your API is authenticated this means that:

- You know how many times members you're about to change are used weekly, daily, etc.
- Most importantly, you know **who** is using these fields the most.

Using this power to communicate changes is a huge benefit of GraphQL, even for internal platforms. A lot of API providers will email integrators when a change is coming. This is a very important practice, but we can push it even further with GraphQL. Because we know who uses every single field, we can tailor our emails to individual integrators that we **know** could be affected.

Email is just one way to communicate changes. This data could be used to build amazing developer dashboards, for example. Here are a few other places you should consider communicating changes:

- Your developer platform's Twitter account.
- Your documentation site.
- A blog post announcing the upcoming changes.
- A change log.

Last Resorts

The sad news is that even with deprecations in place and great communication, chances are a small percentage of your integrators will still be stuck using the deprecated parts of the schema, either because they didn't have the chance to address the change or because they simply did not see your communications.

Any large change will most probably result in some broken integrators, which means you have to weigh your options. Again, if we take Stripe, their API deals with **money**. This heavily skews the versioning strategy towards rarely breaking integrators, and taking the complexity on the provider side of things instead. This is why they've invested in their particular versioning approach.

Using continuous evolution, there's still a last resort we can use to make sure to get the last few integrators moving to the alternatives. API "brownouts" is a technique where we temporarily make a breaking change in the hope that a monitoring system, logs, or a human notices that something has been changed.

Hopefully the error your API is returning will include some kind of information on how to fix it:

```
{
  "errors": [{
    "message":
      "Deprecated: Field `name` does not exist on type `User`.
      Upgrade as soon as possible.
      See: https://my.api.com/blog/deprecation-of-name-on-user"
  }]
}
```

Practically this is usually done using feature toggles. You can enable a brownout flag which would dynamically hide the GraphQL member you're deprecating. This can be done either by using schema visibility techniques or by implementing a custom error in the resolver itself. Take a look at the data you're gathering after some brownout sessions and see if the usage is dropping. You can repeat the experience until you see a drop in usage, or maybe reconsider the deprecation if the usage is just too much.

So, should you version your GraphQL API? That decision ultimately boils down to your own set of tradeoffs, what your clients are expecting, and what kind of expectations you want to set as an API provider. However, more and more, I'm inclined to think that versioning usually ends up causing more trouble than anything, since a lot of the time, there comes a point where changes need to be made, just like in a continuous evolution approach.

GraphQL helps us do continuous evolution in a few ways that make it a bit easier:

- It has first-class deprecation support on fields and most tooling already knows how to use it.
- Additive changes come with no overhead on existing and new clients.
- Usage tracking can be done down to single fields.

These three features make GraphQL a really good candidate for continuous evolution, which is why it is being recommended so strongly. Another thing to keep in mind is that if you opt for a continuous evolution approach first and then decide you absolutely need versioning, that's possible. However, the opposite is much harder.

Still, I feel it's important to mention that continuous evolution can definitely be done in a bad way. It is a big responsibility and (it should not be overdone) can't be abused. That's why additive changes must be the absolute priority before making changes.

Finally, the best way to avoid all these kinds of problems is often at the root: API design. Use a design-first approach with a focus on evolvable design from the get-go. When changes have to be made, we turn to great change management and hope for the best.

Summary

In summary, no versioning approach is perfect and will most likely cause some pain on the server-side and/or the client-side of things. GraphQL has a set of features that makes continuous evolution particularly appealing but versioning a GraphQL is also possible.

If you can, always opt for additive and backward-compatible changes. Use the `@deprecation` directive and don't be afraid to augment its `reason` argument with your own tooling and best practices.

No matter what approach we end up picking, mistakes may happen. It's how we communicate and make these changes that matters most.

Documenting GraphQL APIs

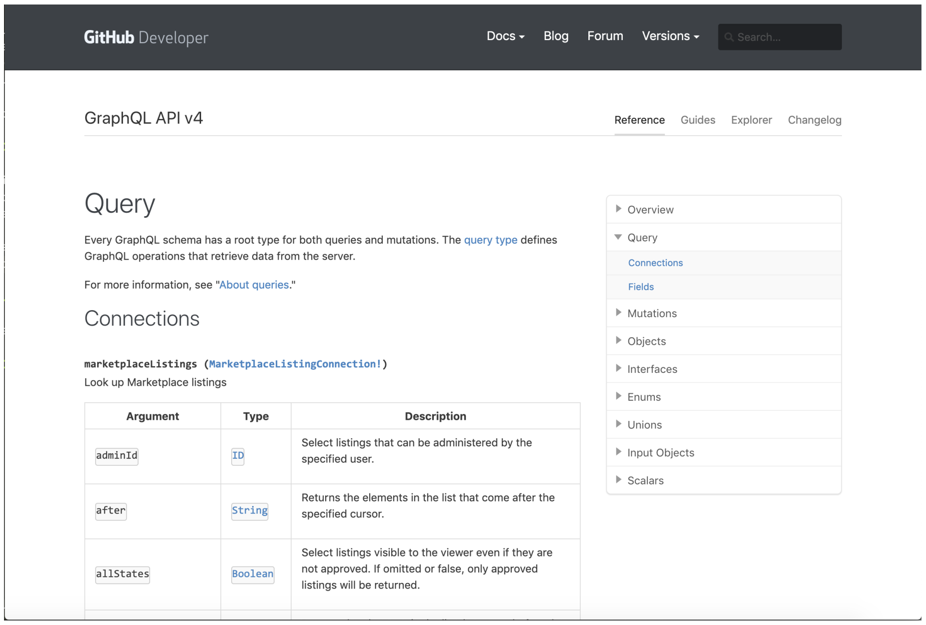
Documentation is the entry point to all new developers wanting to integrate with your API. If we want our APIs to be used and loved, we have no choice but to invest in great documentation. Your documentation doesn't only show potential clients how to achieve things, but what they can achieve using your API. This is a really important point that a lot of providers forget when building developer documentation.

In this chapter, we'll dive into the current state of GraphQL documentation, existing tools, and ways to produce great documentation.

Documentation Generators

Documentation is often listed as a benefit of using GraphQL. In some ways that's very true: the GraphQL type system and introspection capabilities make it easy for clients to discover possibilities. This is what makes GraphQL such a powerful tool to explore GraphQL APIs.

This introspection and type system is something GraphQL API providers used right away to generate documentation. Many tools will use the introspection to generate a complete reference for all types and fields available to clients. This next image is the reference for the `Query` root on GitHub's developer documentation:



GraphQL API v4

Query

Every GraphQL schema has a root type for both queries and mutations. The query type defines GraphQL operations that retrieve data from the server.

For more information, see "About queries."

Connections

`marketplaceListings (MarketplaceListingConnection!)`
Look up Marketplace listings

Argument	Type	Description
`adminId`	`ID`	Select listings that can be administered by the specified user.
`after`	`String`	Returns the elements in the list that come after the specified cursor.
`allStates`	`Boolean`	Select listings visible to the viewer even if they are not approved. If omitted or false, only approved listings will be returned.

Generators are great because they can often be built into your pipeline so that every change to your GraphQL schema gets reflected right away on your documentation site. This solves a common problem with APIs where either the documentation shows wrong information because the implementation has changed, or certain use cases simply never get documented because teams forget.

Whether you use a schema-first approach or a code-first approach, chances are you'll need to filter your schema before using it in some generator tool. The reason being that directives like `@featureFlag`, `@preview`, and `@internal`, or any internal-only information should never be shown in external documentation. If your API is internal-only this might be less of a concern, but it's always a good idea to whitelist the information from introspection before getting it exposed automatically on a docs site. Another approach is to generate two SDL artifact: one that gives a full view of your schema to help internal developers, and one that is intended to be used by documentation tooling.

There are already various ways to generate documentation from a schema. graphql-docs is a great tool we use with success at GitHub. Other tools like graphql-voyager allow you to generate a visual representation for your schema. Over time I'm sure we'll also see more and more SaaS offerings for GraphQL documentation.

The What, Not Just the How

GraphQL APIs have been tagged as having great documentation because of GraphQL's type system and because of how easy it is to generate a reference from introspection, but is it actually all we need? In some ways, the fact that it is so easy to get started might be more of a curse than a blessing. Maybe the best way to understand this is with an example that is often used to describe the benefits of GraphQL, the burger analogy.

Imagine you are at a restaurant and craving for a good hamburger. The analogy is that a typical endpoint-based API would give you a list of pre-made burgers to satisfy common use cases, while GraphQL could simply give you a list of ingredients from which you could make your burger. Initially, this sounds like a great benefit of GraphQL (and it is), but it also illustrates how difficult it can be to document a GraphQL API.

When given a giant list of ingredients, it's very hard for a potential integrator to even understand **what is possible** to achieve with the API. While typical endpoint-based APIs generally group specific use cases in well-defined endpoints, integrators now have to mix and match a number of fields and arguments to build their own resource. The bad news is that most GraphQL documentation currently looks like a bag of fields with no real explanation of what to do with them.

To solve this, we have to focus on the use cases rather than simply generating a reference. A reference is a very useful part of documentation, **but it should not be the only one**. Go check out the documentation website for a GraphQL API you know. Does it talk about what you can achieve with the API or common use cases? Or does it only give you a playground (GraphQL) and a huge reference of types and fields?

My biggest pet peeve with GraphQL documentation is when providers list types by what kind of GraphQL types they are: Here are the Object Types, here are the Input Types, and here are the Union Types. This is absolutely useless for a client who's looking to integrate with your API. Input types are never used in isolation; they're needed as input to a mutation, for example, which returns an object type. What we're interested in is how to add a product to a cart, not the full list of input fields for an `AddProductToCartInput`.

Given these examples, I find it funny when certain people claim that documentation is a big advantage with GraphQL, as compared to REST. The reference might be easier to generate, but the rest of the documentation remains very hard to build. So what can we do?

Workflows and Use Cases

Great GraphQL documentation, and really documentation in general, goes beyond a simple reference and starts with thinking about use cases and common workflows. Let's take an e-commerce API for example. We can think of it in different levels:

1. Types, Fields, Arguments: A simple reference
2. Features / Use Cases: Adding a product to a cart, making a payment for a checkout, canceling an order, etc.
3. Common Applications and Workflows: What can you build with the API? Building checkout extensions, building automation after every order, etc.

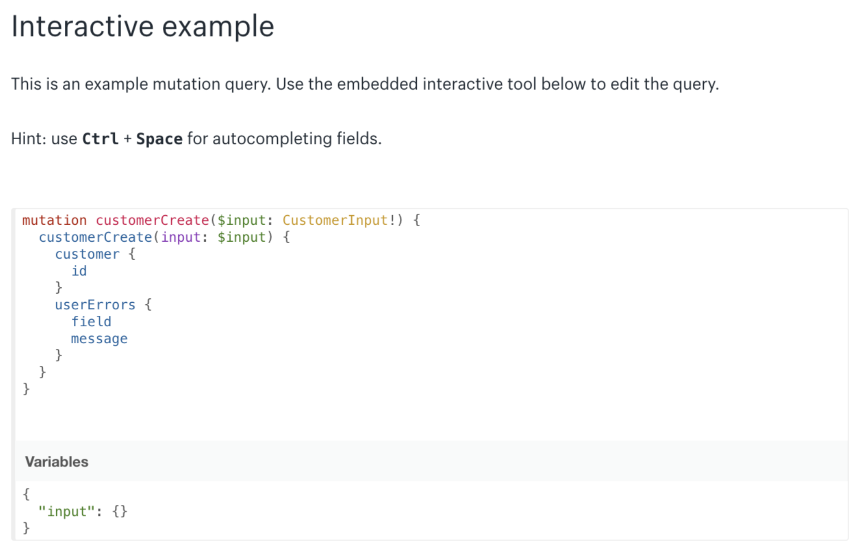
I find the best documentation should include all three levels. A great example of documentation with a focus on typical workflows and possibilities is from Microsoft's Graph API (Not a GraphQL API, confusing I know). On the API's documentation page, we can see a "See what you can do with Microsoft Graph" section which highlights what users may achieve by using the API. This is absolutely great to help onboard clients to your platform, and GraphQL APIs should pay specific attention to this as use cases and possibilities are sometimes lost in a sea of fields and types.

These practices are not easily automated, and that is absolutely ok. Probably the best way to ensure you cover most of these documentation practices is by making sure documentation is not an afterthought. Involve technical writers at the very beginning of your design journey and involve multiple teams and people during the whole process. Think in terms of processes and use cases along the way.

Challenge yourself to go beyond a simple documentation generator for your GraphQL API. Not only will your users be happy, but you will probably improve your API design at the same time.

Example / Pre-Made Queries

A practical way to solve a bit of the “here’s all the ingredients, deal with it” problem is to pre-build queries for integrators. This lets integrators integrate quickly and then tweak queries as needed. An embedded GraphQL is a great way to expose these example queries and lets the users tailor the specific fields down to their needs. Shopify, for example, does a great job at this for the documentation of their mutations for example.


Interactive example

This is an example mutation query. Use the embedded interactive tool below to edit the query.

Hint: use **Ctrl** + **Space** for autocompleting fields.

```
mutation customerCreate($input: CustomerInput!) {
  customerCreate(input: $input) {
    customer {
      id
    }
    userErrors {
      field
      message
    }
  }
}
```

Variables

```
{
  "input": {}
}
```

Changelogs

Changelogs are a great way for API clients to view, or even subscribe to changes in your API. A common approach is to build a list of changes between versions, but they can absolutely be used when opting for continuous evolution as well.

A good example of a GraphQL specific changelog is the GitHub GraphQL API change log, which is generated using graphql-schema_comparator, a tool I wrote for this purpose, which we covered in the tooling and workflows chapter. Shopify and Stripe are two other great examples of API platform changelogs.

Upcoming Changes

Changelogs are great and they can be upgraded by planning upcoming changes, besides logging changes that have already been made. As discussed in the previous chapter on versioning, communication is a big part of a successful breaking change. At GitHub, we built a specific breaking change log for API clients to verify if their integrations could be affected by changes in the future.

We also categorize every change based on its severity: dangerous for changes that could potentially affect an application's runtime behavior, and breaking for schema breaking changes.

Summary

- GraphQL documentation generators are great for API references but lack a human touch.
- Documentation should focus on what is possible with the API, not just how to achieve it.
- Changelogs and upcoming change pages are a great way for clients to keep up with API evolution.

Migrating From Other API Styles

GraphQL being the new kid on the block, I see a lot of teams and companies realizing it fits their needs (or simply jumping on the bandwagon) from an existing API style. This fact has been covered by a lot of posts ever since GraphQL was announced. There are many ways to do it, so let's cover some common situations.

Generators

A very tempting approach to migrating to GraphQL is taking whatever schema or API definition you have from another API style, and try to automate the conversion to a GraphQL schema. For example, openapi-to-graphql is a tool built by a research group at IBM that takes an OpenAPI definition and turns it into a valid GraphQL schema.

I think these can be useful when absolutely needed. However, these tools rarely do an ideal job because different API styles inherently have different design concerns Let's look at one of my favorite examples to see what kind of results these tools can give us.

Let's say we have this OpenAPI definition, which defines a single path and allows clients to publish and unpublish articles using HTTP verbs, a common pattern in HTTP, endpoint-based APIs.

```
paths:
  /articles/{id}/published:
    put: # The put operation publishes an article
    delete: # The delete operation unpublishes an article
```

It's hard for a tool to translate this to a valid GraphQL schema. What we would want with GraphQL would probably look like this instead, a much more RPC oriented approach:

```
type Mutation {
  publishArticle(articleId: ID!): PublishArticlePayload
  unpublishArticle(articleId: ID!): UnpublishArticlePayload
}
```

This is just one example, but you can see how a different API style may require a different design. This is why when given the choice, and if time can be invested, I'd prefer migrating an API to GraphQL by taking a look at the use cases and translating them into a GraphQL interface, rather than translating a REST interface into a GraphQL interface directly.

These tools can get you pretty far, but will always require a human touch if you truly want a well designed "GraphQL-native" API.

REST & GraphQL Alongside

GraphQL is not necessarily there to replace REST, but rather a new way of tackling different challenges in different contexts. This means that providers will not always necessarily replace their REST API with a GraphQL API, and maintaining both, at least for a while, is definitely a possibility. I've had the chance to see multiple approaches to solve this problem.

GraphQL backed by REST

The main way we hear about the transition between REST and GraphQL or maintaining both is to build a GraphQL API on top of an existing REST API. Practically, this means building a GraphQL schema that is resolved by calling down to different REST endpoints. In my experience, this is generally a good pattern as long as performance is kept in check. The same loading issues we saw in the performance chapter can happen when resolvers are calling an endpoint. This could mean terrible performance if implemented naively and could often require building custom batch endpoints if they are not available in your REST API already.

Remember to try and think beyond the current URL structure when building your GraphQL schema. It's very possible your GraphQL types won't be a one-to-one mapping with your REST endpoints, and that's great. It's also possible certain fields and types will be resolved by a combination of REST calls.

REST backed by GraphQL

One other solution that we initially used at GitHub to maintain both our REST and GraphQL APIs was to back REST API endpoints with static GraphQL queries. Since both APIs lived in the same monolith, REST endpoints were modified so that they built a static GraphQL query representing their needs, and executed the query on every request, through the GraphQL engine. Notice that this is not a network call, but simply the REST endpoint calling GraphQL with a pre-defined request, in code.

While this sounds awesome at first, because we're using GraphQL as the source of truth, it comes with certain costs. In fact, I don't recommend this approach anymore. First, there is a strong temptation to design the GraphQL schema so that it fits existing REST endpoints. Just like with the generators, we have to be very careful of not simply copying the design of REST payloads into our GraphQL API, since that makes it easier to use GraphQL from the REST implementation. Not only that, but a very complex dance of error translation will need to happen, for example, translating a GraphQL error to the correct status code.

The other thing to be careful about is that this gives us the impression that we are somehow “reusing” logic by calling down to GraphQL. In an ideal world, REST and GraphQL would not share much since they are both simply an interface. If this technique is tempting to you, maybe it’s because your GraphQL resolvers contain a lot of logic that is tempting to reuse.

So what can we do? Instead of having REST call down to GraphQL, have both REST and GraphQL be independent interfaces and call down to a great domain/business layer! This is already useful when dealing with a single interface like REST, but it becomes crucial when we’re dealing with a UI, a REST API, and a GraphQL API. If your domain logic is not well isolated and reusable, you’ll either centralize it somewhere like a GraphQL resolver, or repeat yourself at all interface layers leading to inconsistencies and bugs. Writing things twice is often a much better idea than opting for a solution that will introduce coupling we don’t want and come bite us later when APIs evolve.

Summary

There are many practical solutions to either migrating or maintaining both GraphQL and REST. The key point here is not to look at your new GraphQL API as a GraphQL version of your REST API, but instead just a new GraphQL interface to your domain and use cases.

GraphQL over REST is a good pattern, but pay attention to the N+1 problem and other performance problems. Don’t copy over REST endpoints to GraphQL, think of your use cases and design the GraphQL API to solve them.

If code reuse is an issue, try to avoid centralizing logic in either your REST or GraphQL API and look for a different layer for this logic. I recommend taking a look at domain driven design for a solution to this.

Closing Thoughts

GraphQL is now here to stay, and for good reasons. We still have a long road ahead, but I hope this journey into what makes an evolvable and reliable GraphQL server will help you support powerful GraphQL platforms. Always remember that the reason we build APIs in the first place is to solve use cases for our audience and clients. Keeping this purpose in mind will ultimately drive your API design and even your implementations. No API style or design is perfect in all contexts. For the situations where GraphQL is the best fit, I really hope this book will have given you tracks that will guide your decisions around building great GraphQL APIs.