Versioning fields in GraphQL

April 15, 2020 · 9 min read

As our application evolves and our users' needs change, the GraphQL API feeding data to it will need to evolve as well, introducing changes to its schema. Whenever the change is non-breaking, as when adding a new type or field, we can apply it directly without fearing side effects. But when the change is a breaking one, we need to make sure we are not introducing bugs or unexpected behavior in the application.

Breaking changes are those that remove a type, field, or directive, or modify the signature of an already existing field (or directive), such as:

- Renaming a field
- Changing the type of an existing field argument, or making it mandatory
- Adding a new mandatory argument to the field
- Adding non-nullable to the response type of a field

In order to deal with breaking changes, there are two main strategies: versioning and evolution, as implemented by REST and GraphQL, respectively.

REST APIs indicate the version of the API to use either on the endpoint URL (such as `api.mycompany.com/v1` or `api-v1.mycompany.com`) or through some header (such as `Accept-version: v1`). Through versioning, breaking changes are added to a new version of the API, and since clients need to explicitly point to the new version of the API, they will be aware of the changes.

GraphQL doesn't dismiss using versioning, but it encourages using evolution. As stated in the GraphQL best practices page:

> *While there's nothing that prevents a GraphQL service from being versioned just like any other REST API, GraphQL takes a strong opinion on avoiding versioning by providing the tools for the continuous evolution of a GraphQL schema.*

Evolution behaves differently in that it is not expected to take place once ever few months, as versioning is. Rather, it's a continuous process, taking place even daily if needed, which makes it more suitable for rapid iteration. This approach has been laid down by Principled GraphQL, a set of best practices to guide the development of a GraphQL service, in its fifth principle:

> *5. **Use an Agile Approach to Schema Development:** The schema should be built incrementally based on actual requirements and evolve smoothly over time*

Evolving the schema

Through evolution, fields with breaking changes must go through the following process:

1. Re-implement the field using a different name.
2. Deprecate the field, requesting clients to use the new field instead.
3. Whenever the field is not used anymore by anyone, remove it from the schema.

Let's see an example. Let's say we have a type `Account`, modeling an account to be a person with a name and a surname through this schema (find an explanation of how schemas are defined here):

```
type Account {
  id: Int
  name: String!
  surname: String!
}
```

In this schema, both the `name` and `surname` fields are mandatory (that's the `!` symbol added after the type `String`) since we expect all people to have both a name and a surname.

Eventually, we also allow organizations to open accounts. Organizations, though, do not have a surname, so we must change the signature of the `surname` field to make it non-mandatory:

```
type Account {
  id: Int
  name: String!
  surname: String # This has changed
}
```

This is a breaking change because the application is not expecting field `surname` to return `null`, so it may not check for this condition, as when executing this JavaScript code:

```
// This will fail when account.surname is null
const upperCaseSurname = account.surname.toUpperCase();
```

The potential bugs resulting from breaking changes can be avoided by evolving the schema:

- We do not modify the signature of the `surname` field; instead, we mark it as deprecated, adding a helpful message indicating the name of the field that replaces it (most likely through a directive `@deprecated` applied on the definition of the field)
- We introduce a new field name `personSurname` (or `accountSurname`) to the schema

Our `Account` type now looks like this:

```
type Account {
  id: Int
  name: String!
  surname: String! @deprecated(reason: "Use `personSurname`")
  personSurname: String
}
```

Finally, by collecting logs of the queries from our clients, we can analyze whether they have made the switch to the new field. Whenever we notice that the field `surname` is no longer used by anyone, we can then remove it from the schema:

```
type Account {
  id: Int
  name: String!
  personSurname: String
}
```

Issues with evolution

The example described above is very simple, but it already demonstrates a couple of potential problems from evolving the schema:

Problem	Description
Field names become less neat	The first time we name the field, we will possibly find the optimal name for it, such as `surname`. When we need to replace it, though, we will need to create a different name for it that may be suboptimal (the optimal is already taken!). All possible replacements in the example above have problems: • `personName` makes explicit that the account is for a person, so if, later on, we must open an account for a non-person with a surname (I don't know... a Martian?), then we will need to evolve the schema again so as to keep consistent names • The "account" bit in `accountName` is completely redundant since the type is already `Account` • Otherwise, what other name to use? `surname1`? `surnameNew`? Or even worse, `surnameV2`? As a consequence, the updated schema will be less understandable and more verbose.
The schema may accumulate deprecated fields	Deprecating fields is most sensible as a temporary circumstance; eventually, we would really like to remove those fields from the schema to clean it up before they start accumulating. However, there may be clients that don't revise their queries and still fetch information from the deprecated field. In this case, our schema will slowly but steadily become a kind of field cemetery, accumulating several different fields for the same functionality.

Is there any way we can solve these issues?

Versioning fields

There are two different approaches for coding a GraphQL schema: code-first and schema-first, as I explained in a previous article. In the examples above, I used the schema-first approach to define the schema, based on the Schema Definition Language (SDL).

When defining the schema through the SDL, it seems to me there is no way to overcome the issues I just mentioned because the contract of the schema is explicit. And since the new contract is different (a different field name, an additional `!` symbol to make its type mandatory, etc.), there is no alternative than to write it again, in its new form, keeping the old form, too.

Using the code-first approach, though, the schema can become dynamic, changing its shape and attributes depending on the context. In my previous article "Speeding up changes to the GraphQL schema," I described how this strategy works, with the goal of allowing different teams in the company to provide their own implementation for a generic field `discountedPrice` on a type `Product`.

The strategy is to enable more than one resolver to satisfy a field. The implementing resolver is chosen on runtime based on contextual information (such as the type of the product, the current date, or others) by asking each resolver in a chain whether it can process the query, until either we find a resolver that does or we reach the end of the chain.

We can apply this same strategy to version fields! We can create our field with an argument called `version`, through which we specify which version of the field to use.

In this scenario, we will still have to keep the implementation for the deprecated field, so we are not improving in that concern. However, its contract becomes hidden: the new field can now keep its original name (there is no need to rename it from `surname` to `personSurname`), preventing our schema from becoming too verbose.

Please note that this concept of versioning is different than that in REST:

- REST establishes an all-or-nothing situation in which the whole queried API has the same version since the version to use is part of the endpoint
- In this other approach, each field is versioned independently

Hence, we can access different versions for different fields, like this:

```
query GetPosts {
  posts(version: "1.0.0") {
    id
    title(version: "2.1.1")
    url
    author {
      id
      name(version: "1.5.3")
    }
  }
}
```

Moreover, by relying on semantic versioning, we can use the version constraints to choose the version, following the same rules used by npm for declaring package dependencies. Then, we rename field argument `version` to `versionConstraint` and update the query:

```
query GetPosts {
  posts(versionConstraint: "^1.0") {
    id
    title(versionConstraint: ">=2.1")
    url
    author {
      id
      name(versionConstraint: "~1.5.3")
    }
  }
}
```

Applying this strategy to our deprecated field `surname`, we can now tag the deprecated implementation as version `"1.0.0"` and the new implementation as version `"2.0.0"` and access both, even on the same query:

```
query GetSurname {
  account(id: 1) {
    oldVersion:surname(versionConstraint: "^1.0")
    newVersion:surname(versionConstraint: "^2.0")
  }
}
```

I've implemented this solution on my own GraphQL server, GraphQL by PoP. Check out the response to this query on GraphiQL:

Querying fields through version constraints.

Defining the default version for a field

What should happen when we do not specify the `versionConstraint` argument? For instance, to which version should field `surname` in the query below resolve to?

```
query GetSurname {
  account(id: 1) {
    # Which version should be used? 1.0.0? 2.0.0?
    surname
  }
}
```

We have two concerns here:

1. Deciding which is the default version to use when none is provided
2. Informing the client that there are several versions to choose from

Let's tackle these concerns next. But before that, we need to find out how well GraphQL provides contextual feedback when running a query.

Providing contextual feedback when running queries

I need to point out a less-than-ideal circumstance with GraphQL right now: it doesn't offer good contextual information when running queries. This is evident concerning deprecations, where deprecation data is shown only through introspection by querying fields `isDeprecated` and `deprecationReason` on the `Field` and `Enum` types:

```
{
  __type(name: "Account") {
    name
    fields {
      name
      isDeprecated
      deprecationReason
    }
  }
}
```

The response will be:

```
{
  "data": {
    "__type": {
      "name": "Account",
      "fields": [
        {
          "name": "id",
          "isDeprecated": false,
          "deprecationReason": null
        },
        {
          "name": "name",
          "isDeprecated": false,
          "deprecationReason": null
        },
        {
          "name": "surname",
          "isDeprecated": true,
          "deprecationReason": "Use `personSurname`"
        },
```


However, when running a query involving a deprecated field...

```
query GetSurname {
  account(id: 1) {
    surname
  }
}
```

...the deprecation information will not appear in the response:


```
{
  "data": {
    "account": {
      "surname": "Owens"
    }
  }
}
```

This means that the developer executing the query must actively execute introspection queries to find out whether the schema was upgraded and any field deprecated. That may happen... once in a long while? Quite possibly never?

It would be a great improvement towards revising outdated queries if the GraphQL API provided deprecation information when executing queries that involve deprecated fields. This information may ideally be given under a new top-level entry `deprecations`, appearing after `errors` and before `data` (following the spec's suggestion for the response format).

Since a `deprecations` top-level entry is not part of the spec, GraphQL server implementors can still add support for better feedback by using the wildcard top-level entry `extensions`, which allows extending the protocol as needed. In GraphQL by PoP, the response to this query on GraphiQL looks like this:

Deprecation information on the query response.

Publicizing versions through warnings

We have just learnt that the GraphQL server can use the `extensions` top-level entry to provide deprecations. We can use this same methodology for adding a `warnings` entry, in which we inform the developer that a field has been versioned. We do not provide this information always; only when the query involves a field which has been versioned, and the `versionConstraint` argument is absent.

We can now create strategies for the default version for a field.

Strategies for versioning

There are several approaches we can employ, including:

1. Make `versionConstraint` mandatory
2. Use the old version by default until a certain date, on which the new version becomes the default

3. Use the latest version by default and encourage the query developers to explicitly state which version to use

Let's explore each of these strategies and see their responses when running this query:

```
query GetSurname {
  account(id: 1) {
    surname
  }
}
```

1. Make `versionConstraint` mandatory

This is the most obvious one: forbid the client from not specifying the version constraint by making the field argument mandatory. Then, whenever not provided, the query will return an error.

Running the query will respond with:

```
{
  "errors": [
    {
      "message": "Argument 'versionConstraint' in field 'surname' cannot be
empty"
    }
  ],
  "data": {
    "account": {
      "surname": null
    }
  }
}
```

2. Use the old version by default until a certain date, on which the new version becomes the default

Keep using the old version until a certain date, when the new version will become the default. While in this transition period, ask the query developers to explicitly add a version constraint to the old version before that date through the new `extensions.warnings` entry in the query.

Running the query will respond with:


```
{
  "extensions": {
    "warnings": [
      {
        "message": "Field 'surname' has a new version: '2.0.0'. This version
will become the default one on January 1st. We advise you to use this new
version already and test that it works fine; if you find any problem, please
report the issue in https://github.com/mycompany/myproject/issues. To do the
switch, please add the 'versionConstraint' field argument to your query
(using npm's semver constraint rules; see https://docs.npmjs.com/about-
semantic-versioning): surname(versionConstraint:\"^2.0\"). If you are unable
to switch to the new version, please make sure to explicitly point to the
current version '1.0.0' before January 1st:
surname(versionConstraint:\"^1.0\"). In case of doubt, please contact us at
name@company.com.",
    ]
  },
  "data": {
    "account": {
      "surname": "Owens"
```


3. Use the latest version and encourage the users to explicitly state which version to use

Use the latest version of the field whenever the `versionConstraint` is not set, and encourage the query developers to explicitly define which version must be used, showing the list of all available versions for that field through a new `extensions.warnings` entry:

Running the query will respond with:

```
{
  "extensions": {
    "warnings": [
      {
        "message": "Field 'surname' has more than 1 version. Please add the
'versionConstraint' field argument to your query to indicate which version to
use (using npm's semver constraint rules; see https://docs.npmjs.com/about-
semantic-versioning). To use the latest version, use:
surname(versionConstraint:\"^2.0\"). Available versions: '2.0.0', '1.0.0'.",
    ]
  },
  "data": {
    "account": {
      "surname": "Owens"
    }
  }
}
```

Versioning directives

Since directives also receive arguments, we can implement exactly the same methodology to version directives, too!

For instance, when running this query:

```
query {
  post(id: 1) {
    oldVersion:title@makeTitle(versionConstraint: "^0.1")
    newVersion:title@makeTitle(versionConstraint: "^0.2")
  }
}
```

It produces a different response for each version of the directive:

Querying a versioned directive.

And when running the query without providing the version constraint:

```
query {
  post(id: 1) {
    title@makeTitle
  }
}
```

It assumes a default version to use and produces a warning message for the developer to revise the query:

> ***Querying a versioned directive without version constraints.***

Conclusion

You may be able to find articles on the internet comparing GraphQL and REST on all their characteristics and musing about which is better. When it comes to comparing versioning and evolution specifically, the response (as usual) is: "Neither is particularly better; they both have advantages and disadvantages, and they are more or less suitable depending on the context."

What is better, though, is to be able to combine both approaches and obtain the best of both worlds: an API that is itself version-less and is upgraded through evolution, but having fields (and directives) that can be versioned. Please notice that there is no contradiction here — having versioned fields (and directives) doesn't make the API versioned since each field is still independently queried.

In this article, we learned how this strategy works and how it can be implemented. To be honest, I do not know how many GraphQL servers out there currently support fully implementing this strategy, since it involves sending custom data under the top-level `extensions` entry in the response when executing a query.

However, even without this enhanced feedback information, we can version the fields (and directives) in our schema so as to avoid naming fields like `personSurname`.

Keep reading more

This article is part of an ongoing series on conceptualizing, designing and implementing a GraphQL server. The previous articles from the series are:

1. Designing a GraphQL server for optimal performance
2. Simplifying the GraphQL data model
3. Schema-first vs. code-first development in GraphQL
4. Speeding up changes to the GraphQL schema

Leonardo Losoviz Follow

Freelance developer and writer, with an ongoing quest to integrate innovative paradigms into existing PHP frameworks, and unifying all of them into a single mental model.