PRODUCTION READY GRAPHQL

# GraphQL Continuous Evolution Guide

# Introduction

Thank you for purchasing Production Ready GraphQL.

Hard versioning can sometimes be avoided with GraphQL by practicing careful continuous evolution of our APIs.

This guide compiles everything you need to know about making changes in a GraphQL schema and how to approach them.

- Marc-Andre

## Schema Evolution

Not all changes are created equal. In this section, we'll analyze a few different types of changes and what makes them safe or unsafe.

**Safe**

*Safe or non-breaking changes can be made on existing GraphQL schemas at any time and are very unlikely to have any effect on existing clients.*

**Dangerous**

*Dangerous changes are changes that are in theory non-breaking but can affect clients in subtle ways. Consider communicating those changes to help client-side developers modify their applications if needed.*

**Breaking**

*Breaking changes are changes that will almost certainly result in the breakage of client applications. They should be done rarely and always be communicated beforehand.*

## Additive Changes

**Most additive changes in GraphQL are safe.** For example, adding fields & adding types is unlikely to cause any issues for clients. In this section, we'll cover some additive changes that are trickier to handle.

### Adding Optional Arguments & Input Fields Dangerous

Adding optional arguments is generally safe since clients are not forced into providing it. However, be careful that the resolver does not behave differently when the value is not provided. Not passing the argument should provide the same result as before.

```
type Query {
-  users(first: Int!): String!
+  users(first: Int!, showArchived: Boolean): String!
}
```

Default values can help with preserving the default behavior.

```
type Query {
-  users(first: Int!): String!
+  users(first: Int!, showArchived: Boolean = False): String!
}
```

### Adding Required Arguments & Input Fields Breaking

Adding the required arguments is always a breaking change unless a default value is also provided.

```
type Query {
-  users(first: Int!): String!
+  users(first: Int!, showArchived: Boolean!): String!
}
```

## Adding New Interface Implementations Dangerous

While adding types is a safe change, adding a new type that implements an existing interface or implementing an interface on an existing type may cause clients to behave differently. This change should be approached carefully.

```diff
- type Bot {
+ type Bot implements Actor {
  name: String!
}
```

When adding new implementations, you should make sure that client queries selecting for an `Actor` would not get broken by the addition of a new `User` type. Consider communicating this change ahead of time.

```graphql
query GetActor {
  actor {
    __typename

    ... on User {
      login
    }
    ... on Team {
      name
    }
  }
```

Clients should at the very least handle new concrete types with a proper `else` clause if possible.

```
switch(actor.__typename) {
  case “User”:
    actor.login
    break;
  case “Team”:
    actor.name
    break;
  default:
    `Actor with ID: ${actor.id}`
}
```

## Adding New Union Members Dangerous

Just like with adding interface implementations, adding new union types to unions may cause problems for clients at runtime. In a way, they are even more dangerous than adding interface implementations since they share no common contract.

```
- union SearchResult = Post | User
+ union SearchResult = Post | User | Picture
```

Consider documenting how clients should behave with unknown type names in the description of the type, and announce these changes if the new value absolutely needs to be handled by clients.

## Adding New Enum Values Dangerous

Similar to the two last changes, adding enum values can potentially impact client-side logic at runtime.

```
enum OrderStatus {
  PENDING
  COMPLETED
+ CANCELED
}
```

## Modifications

**Almost all modifications in place to a schema are breaking.** For example, changing a field's type or changing the name of a type is a big breaking change.

### Changing Default Values Dangerous

Changing the default value of an argument or input field is unlikely to be a breaking change in terms of the schema itself, but is very likely to cause issues at runtime if the default value affects the runtime behavior of the field.

```
type Query {
+  users(first: Int!, showArchived: Boolean = True): String!
+  users(first: Int!, showArchived: Boolean = False): String!
}
```

Avoid this change in general, but it may be possible to achieve if the behavior of the field does not change.

### Changing Descriptions Safe

Changing the description of fields, types and any member is unlikely to cause any harm to clients. Clients should not depend on schema descriptions for runtime logic!

## Subtractive Changes

**All subtractive changes to a schema are breaking changes.** Removing fields, types, enum values, interface implementations, directives, and union members are always breaking unless of course there is no usage of those members. Always make these changes with plenty of communication beforehand, and try to represent them as additive changes instead if possible.

In the next section, we'll see how to make these changes progressively and safely!

# Making Changes in Practice

Although we try to avoid breaking changes at all cost they are sometimes unavoidable. Here's how to approach some common changes to a schema.

## Removing fields

Removing a field is a breaking change. Common reasons for removing a field are security, performance, changing domain, or naming issues.

```
type Query {
-  users(first: Int!): String!
}
```

The best way to remove a field is by introducing its replacement and deprecating it as a first step.

```
type Query {
-  users(first: Int!): [User!]!
+  users(first: Int!): [User!]! @deprecated(reason: “Users is deprecated and
is getting replaced by the field `topUsers`.”)
+  topUsers(first: Int!): [User!]!
}
```

The old field may be removed after a certain period and if the usage for it has gone down. Keep in mind you don't necessarily have to make the change unless absolutely needed. Additive changes and deprecations are sometimes enough to keep evolving the API.

```
type Query {
-  users(first: Int!): [User!]! @deprecated(reason: “Users is deprecated and
is getting replaced by the field `topUsers`.”)
   topUsers(first: Int!): [User!]!
}
```

## Changing Arguments

Removing or modifying arguments is tricky because arguments cannot be marked as deprecated per the spec. There are a few ways to approach the change.

```diff
type Query {
-  users(first: Int!, adminsOnly: Boolean): String!
+  users(first: Int!): String!
}
```

Instead of trying to remove or change arguments, it's often more simple to expose a new field without that change and deprecating the old field.

```diff
type Query {
-  users(first: Int!, adminsOnly: Boolean): String!
+  users(first: Int!, adminsOnly: Boolean): String! @deprecated(reason: "users
is deprecated, use `users` for users and `admins` to get users that are ad-
mins"
+  allUsers(first: Int!): String!
+  admins(first: Int!): String!
}
```

If you need to make a change to an existing field, because arguments can't be deprecated just yet, you should indicate that the argument is deprecated through its description.

```diff
type Query {
-  users(first: Int!, adminsOnly: Boolean): String!
+  users(
     first: Int!,
     # DEPRECATED: This argument will be removed. Use field `admins`.
+    adminsOnly: Boolean
+  ): String!
}
```

## Non-Null to Null

Changing a field from non-null to null is one of the most annoying changes to make. This is part of why I recommend thinking twice before making a field non-null.

For scalar fields, you might be able to save your users from errors by returning the empty value instead of null.

```
const nameResolver = (obj, args, ctx) ⇒ { user.name || “” };
```

For object types, sometimes it’s possible to use a “Null Object” when the result is null.

```
const authorResolver = (obj, args, ctx) ⇒ {
  if (obj.author) {
    return obj.author;
  } else {
    return new NullAuthor();
  }
};
```

### Removing an entire type

Sometimes we want to deprecate a full object type. Object types can't be deprecated directly. Deprecating an object type or an input type is **equivalent to deprecating all fields, arguments and input fields of that type.**

```
type User {
  name: String!
}

+ type Author {
+   name: String!
+ }

type Post {
  author: User! @deprecated(reason: “Type `user` is deprecated. User 
`postAuthor` instead”)
+ postAuthor: Author!
}

type Query {
  user(id: ID!): User @deprecated(reason: “Type `user` is deprecated, use 
`Query.author` instead.”)
+ author: Author!
}
```

Note that you might want to deprecate using that type **within your codebase** to avoid developers to use that `User` type for new fields.

Removing a type is even trickier when it's part of union types or implements interfaces. Once again, union members and interface implementations cannot be marked as deprecated. This means that fields like `node` may stop working correctly if the type you're removing was reachable through that field.

Your best bet in these cases are to either keep this type as part of unions and through interfaces or to communicate that change very carefully through descriptions and out of band communication like documentation and emails.

## Changing a field’s type

Changing a field’s type is not a change we can make easily. Once again, approaching the change in an additive is often your best bet.

```
type User {
  bestFriend: String! @deprecated(reason: “Use `bestFriendObject` instead.”)
+ bestFriendObject: User!
}
```

As you might have noticed, the downside of additive changes is that often the best names are already taken. If wanted, you may remove the original field and reintroduce it under the new object at that point.

```
type User {
- bestFriend: String! @deprecated(reason: “Use `bestFriendObject` instead.”)
  bestFriendObject: User!
+ bestFriend: User!
}
```

## Deprecations do not necessarily mean removals

Keep in mind that when we deprecate schema members, we don’t necessarily need to have a plan to fully remove them. By deprecating, we already discourage any new usage, and often these members get hidden from GraphQL tooling.

Unless you absolutely need the deprecated schema member to go away, I suggest you be patient and wait for usage to subside before removing fields.

## Schema Evolution at Scale Check List

A handy checklist to follow to make continuous evolution of a GraphQL API a painless process. Feel free to add this to your internal documentation.

- [ ] Avoid unwanted breaking changes by using a tool like xuorig/graphql-schema_comparator
- [ ] Abstract away GraphQL deprecations with a deprecated helper function

```
class PostType < BaseObjectType
  description "A blog post"
  field :id, ID, null: false

  field :title, String, null: false do
     deprecated(
       reason: “Title will be replaced by titleWithDescription”,
       new_field: “PostType.titleWithDescription”,
       sunset: “2020-12-01”
     )
  end
end
```

This helps to standardize deprecation messages and encode more useful information for users

- [ ] Use descriptions to encode upcoming changes on arguments and input values until the @deprecated directive gets added to the specification. Use your custom deprecated

```
type Query {
  users(
    first: Int!,
    # DEPRECATED: This argument will be removed. Use field `admins`.
    adminsOnly: Boolean
  ): String!
}
```

- [ ] Add deprecation warnings to documentation. consider making deprecated members hidden or at least of lesser importance than others.

- [ ] Make sure you identify clients through an app or client ID, and track their usage over time. Ask for an email if possible.

- [ ] Start contacting clients using the deprecated members as soon as you deprecate a field. Leave plenty of time to make the change. Include the exact queries they are making. Bonus points if you can include an equivalent query that they can make instead. Possible with great tracking of queries.

- [ ] Use a `warnings` extension to warn developers using deprecated features.

```
{
  “data”: { … },
  "extensions": {
    “warnings”: [{
      “message”: “Field `User.login`. is deprecated. Please use 
`User.name` instead.”,
    }]
  }
}
```

- [ ] Last resort: Use “brownouts” (Temporarily disabling deprecated schema members) and follow up on data to see if usage is going down. This can be done by throwing an error within a resolver or using schema visibility filters. Include information about the

```
const loginResolver = (obj, args, ctx) ⇒ {
  throw new DeprecatedFieldError(`
    This field will soon be removed in favor of `User.name`.
    See https://api.com/blog/deprecation-2020-12-01 for more info')
  `)
};
```