The Case Against __typename

GraphQL Conf 2026

Sabrina Wasserman
Software Engineer

Overview

01 What is `__typename`, and why do we need it?

02 How does `__typename` fall short?

01 What is `__typename`?

What is `__typename`?

> *"The* __typename *field is a special meta-field that automatically exists on every Object type and resolves to the name of that type, providing a way to differentiate between data types on the client."*

graphql.org/learn/schema/

What is __typename?

```
type Contact  {
  display_name : String
}


query ContactsQuery($query: String!) {
  contacts(query:$query) {
    __typename
    display_name
}
```

```
{
  "data" : {
    "contacts": [
      {
        "__typename": "Contact",
        "display_name” : "Ada Lovelace"
      },
      {
        "__typename": "Contact",
        "display_name” : "Alan Turing"
      }
    ]
  }
}
```

```
interface Contact  {
  display_name : String
}

type Company implements Contact  {
  display_name: String
}

type Person implements Contact  {
  display_name: String
  birthday: Date
}

query ContactsQuery($query: String!) {
  contacts(query:$query) {
    __typename
    display_name
}
```

```
{
  "data" : {
    "contacts": [
      {
        "__typename": "Company",
        "display_name" : "Ada Lovelace Co."

      },
      {
        "__typename": "Person",
        "display_name" : "Alan Turing"

      }
    ]
  }
}
```

Put simply, `__typename` is:

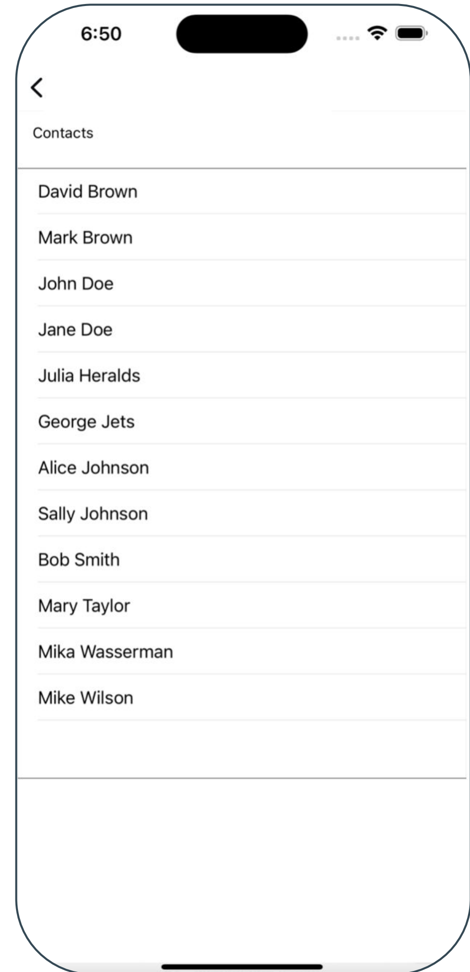
- A **String** field, available on all objects
- A quick-and-dirty way to check the **concrete type of an object at runtime**

02 How Does `__typename` Fall Short?

Building a Contacts Book

Suppose I want to make a simple searchable contacts directory for a small business, where I can lookup and display the names of matching results from my search query.

In my contact book, I can have both **companies,** and **individual people**



We can easily model the schema as:

```
interface Contact  {
  display_name : String
}

type Company implements Contact  {
  display_name : String
}

type Person implements Contact  {
  display_name : String
  birthday: Date
}
```

And query for the data:

```
query ContactsQuery($query: String!) {
  contacts(query: $query) {
    __typename
    display_name

   ... on Person {
      birthday
   }
}
```


```
{
  "data" : {
    "contacts": [
      {
        "__typename": "Company",
        "display_name” : "Ada Lovelace Co."

      },
      {
        "__typename": "Person",
        "display_name” : "Alan Turing",
        "birthday” : "1912-06-23"
      }
    ]
  }
}
```

and display companies and contacts differently depending on `__typename`

```
if (model.typeName == “Company”) {
   displayCompany(model.displayName)
}
else if (model.typeName == “Person”) {
   displayPerson(model.displayName, model.birthday)
}
```

Problem 1 - Forwards Compatibility

Problem 1.1: Schema Upgrades

- Oftentimes, the initial schema we write needs to get more complex as the product matures
- A **very common example** - upgrading a type from a concrete to interface type

```
type Person implements Contact {
  display_name : String
  birthday: Date
}
```

```
interface Person implements Contact {
  display_name : String
  birthday: Date
}

type Employee implements Person  {
  display_name : String
  birthday: Date
  employee_id: ID
}

type Customer implements Person  {
  display_name : String
  birthday: Date
}
```

For older clients, our query hasn't changed, nor has the client logic:

```
query ContactsQuery($query: String!) {
  contacts(query: $query) {
    __typename
    display_name

   ... on Person {
      birthday
   }
}
```

```
if (model.typeName == "Company") {
   displayCompany(model.displayName)
}
else if (model.typeName == "Person") {
   displayPerson(model.displayName, model.birthday)
}
```

But, **the network response has:**

```
{
  "data" : {
    "contacts": [
      {
        "__typename": "Company",
        "name” : "Ada Lovelace Co."

    },
      {
        "__typename": "Person",
        "name” : "Alan Turing",
        "birthday” : "1912-06-23"
      }
    ]
  }
}
```

```
{
  "data" : {
    "contacts": [
      {
        "__typename": "Company",
        "display_name” : "Ada Lovelace Co."

    },
      {
        "__typename": "Customer",
        "display_name” : "Alan Turing",
        "birthday” : "1912-06-23"
      }
    ]
  }
}
```

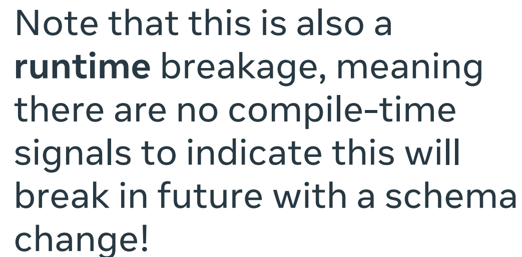
```
if (model.typeName == “Company”) {
   displayCompany(model.displayName)
}
else if (model.typeName == “Person”) {
   displayPerson(model.displayName, model.birthday)
}
```

`Person` is no longer a concrete type, so this check fails!

But we still want future subtypes of `Person` to work here!

Importantly, **the model is still is a Person**, but `__typename` no longer gives us that info!

```
if (model.typeName == “Company”) {
   displayCompany(model.displayName)
}
else if (model.typeName == “Person”) {
   displayPerson(model.displayName, model.birthday)
}
```



Note that this is also a **runtime** breakage, meaning there are no compile-time signals to indicate this will break in future with a schema change!

```
if (model.typeName == “Company”) {
   displayCompany(model.displayName)
}
else if (model.typeName == “Employee” ||
         model.typeName == “Customer”) {
   displayPerson(model.displayName, model.birthday)
}
```

Even if we fix this going forward on the client it still:

1. Breaks for old clients
2. Is not robust against further schema changes!

Problem 1.2: Adding A Sub-Type

- Similar to upgrading a type, **adding a sub-type of an interface** also presents the same issue
- Even if the client logic is updated to check for sub-types, new subtypes still break old clients
- For our contact book, suppose we want to add a new type of person, `Contractor`

```
type Contractor implements Person {
  display_name : String
  contract_active: Boolean
}
```

Problem 1.2: Adding A Sub-Type

Now, our subtype logic on older clients is no longer exhaustive, so any returned contractors do not properly display as a `Person` type!

```
if (model.typeName == “Company”) {
   displayCompany(model.displayName)
}
else if (model.typeName == “Employee” ||
         model.typeName == “Customer”) {
   displayPerson(
     model.displayName
     model.birthday)
}
```

```
{
  "data" : {
    "contacts": [
      {
        "__typename": "Company",
        "display_name” : "Ada Lovelace Co."
      },
      {
        "__typename": "Contractor",
        "display_name” : "My Fave Contractor"
      }
    ]
  }
}
```

This isn’t just a problem for product code…

```
Class Contact {

   Company getCompany() {
     if (self.typeName == “Company”) {
       return reinterpretAsCompany(self);
     }
   }

   Person getPerson() {
     if (self.typeName == “Person”) {
       return reinterpretAsPerson(self);
     }
   }
}
```

- If we want to provide **type-safe generated GraphQL models**, we want a mechanism to check type before returning the model to the client
- Checking `__typename` here presents the same forwards compatibility concerns as in product code!

In both of these cases, **older clients are capable of handling the interface types,** but fail checks by depending on `__typename`!

But this is still what many clients do!

From the Relay documentation:

TIP

If you need to do something totally different depending on what type it is, you can select a field called `__typename`, which returns a string with the name of the concrete type that you got (e.g., `"Person"` or `"Organization"`). This is a built-in feature of GraphQL.

What Can We Do To Resolve This?

- One frequent solution to this issue is to **never allow changes in your schema that would cause `__typename` to change**, even though by definition these changes **are not breaking in the spec**

- In practise, this frequently means adding a v2 of a field with a new type, and annotating the previous field with `@deprecated`
  - This means both the old and new versions need to stay around in order to still maintain older clients
  - Ideally, we want a solution to this which **allows us the flexibility to change our schema without breaking or force-upgrading old clients.**

Problem 2 - Nested Abstract Types

Nested Abstract Types

- Nested abstract types are particularly painful to manage, because `__typename` only provides the **concrete object type**; it supplies no information about whether or not the object implements a given interface

- This is particularly painful for nested abstract types, because we lose all type information for the entire path of data

Nested Abstract Types

Suppose both Company and Person implement types for their legal names

```
interface Contact  {
  display_name : String
}

type Company implements Contact  {
  display_name : String
  legal_name : String
}

interface Person implements Contact {
  display_name : String
  birthday: Date
  legal_name: PersonLegalName
}
```

```
type PersonLegalName {
  first_name : String
  last_name: String
}

type Employee implements Person  {
  display_name : String
  birthday : Date
  employee_id : ID
  legal_name: PersonLegalName
}

type Customer implements Person  {
  display_name : String
  birthday: Date
  legal_name: PersonLegalName
}
```

We can query for legal name to display depending on the type in the query

```
query Search($query: String!) {
  contacts(query: $query) {
    __typename
    display_name
    ... on Person {
      legal_name {
        first_name
        last_name
      }
    }
    ... on Company {
      legal_name
    }
}
```

```
{
  "data" : {
    "contacts": [      {
        "__typename": "Customer",
        "display_name” : "Alan Turing",
        "legal_name” : {
          "first_name” : "Alan",
          "last_name” : "Turing",
          }
      },
      {
        "__typename": "Company",
        "display_name” : "Ada Lovelace Co.",
        "legal_name” : "Lovelace Co."
      }
    ]
  }
}
```

- At runtime, we need to know how to handle `legal_name` in order to properly parse and display it
- Ideally, we’d use `__typename`, but it doesn’t encode any information about whether or not `Customer` is of type `Person`!

```
if (model.typeName == “Company”) {
   displayLegalName(model.legalName)
}
else if (model.typeName == “Employee” ||
         model.typeName == “Customer”) {
   displayLegalPersonName(
     model.legalName.firstName,
     model.legalName.lastName)
}
```

Attempting to use `__typename` to solve this brings us back to the same non-forward compatible mess as before!

```
if (model.legalName is String) {
   displayLegalName(model.legalName)
}
else if (model.legalName is Object) {
   displayLegalPersonName(
     model.legalName.firstName,
     model.legalName.lastName)
}
```

Attempting to check the type of `legal_name` opens a whole can of worms of complexity.

What if `Company`'s type for `legal_name` is also an object?

What is `Company`, `Employee`, and `Customer` all have their own different types for `legal_name`?

These problems boil down to the fact that the way we ***write*** GraphQL queries is not analogous to how we ***receive*** GraphQL data back from the network

- Namely, all fragment spreads and type switches are lost in the collapsed response

```
query ContactsQuery($query: String!) {
  contacts(first: 1, query: $query) {
    __typename
    display_name
    ... on Person {
      birthday
      ... on Employee {
         employee_id
      }
    }
    ... on Company {
      legal_name
   }
}
```

```
{
  "data" : {
    "contacts": [
      {
        "__typename": "Employee",
        "display_name” : "Ada Lovelace",
        "employee_id” : "1"

       },
    ]
}
```

We need some mechanism that:

1. Gives us typing information that doesn't break if `__typename` changes
2. Answers the question "does this object implement this interface"

Does where we select __typename matter?

```
query ContactsQuery($query: String!) {
  contacts(first: 1, query: $query) {
    __typename
    display_name
    ... on Person {
      birthday
      ... on Employee {
         employee_id
      }
    }
    ... on Company {
      legal_name
   }
}
```

```
{
  "data" : {
    "contacts": [
      {
        "__typename": "Company",
        "display_name" : "Ada Lovelace Co.",
        "legal_name" :  "Lovelace Co.",
      },
      {
        "__typename": "Employee",
        "display_name" : "Sabrina Waserman",
        "birthday" : "1912-06-23"
        "employee_id" : "1",

      },
      {
        "__typename": "Customer",
        "display_name" : "Alan Turing",
        "birthday" : "1912-06-23"
      }
    ]
  }
}
```

__typename no longer returns for Company type!

```
query ContactsQuery($query: String!) {
  contacts(first: 1, query: $query) {
    display_name
    ... on Person {
      __typename
      birthday
      ... on Employee {
        employee_id
      }
    }
    ... on Company {
      legal_name
    }
  }
}
```

```
{
  "data" : {
    "contacts": [
      {
        "display_name" : "Ada Lovelace Co.",
        "legal_name" :  "Lovelace Co.",
      },
      {
        "__typename": "Employee",
        "display_name" : "Sabrina Waserman",
        "birthday" : "1912-06-23"
        "employee_id" : "1",

      },
      {
        "__typename": "Customer",
        "display_name" : "Alan Turing",
        "birthday" : "1912-06-23"
      }
    ]
  }
}
```

`__typename` no longer returns for `Person` or `Company` type!

```
query ContactsQuery($query: String!) {
  contacts(first: 1, query: $query) {
    display_name
    ... on Person {
      birthday
      ... on Employee {
         __typename
         employee_id
      }
    }
    ... on Company {
      legal_name
   }
}
```

```
{
  "data" : {
    "contacts": [
      {
        "display_name” : "Ada Lovelace Co.",
        “legal_name” :  "Lovelace Co.",
      },
      {
        "__typename": "Employee",
        "display_name” : "Sabrina Waserman",
        "birthday” : “1912-06-23”
        "employee_id” : "1",

      },
      {
        "display_name” : "Alan Turing",
        "birthday” : “1912-06-23”
      }
    ]
  }
}
```

__typename no longer returns for any Person!

```
query ContactsQuery($query: String!) {
  contacts(first: 1, query: $query) {
    display_name
    ... on Person {
      birthday
      ... on Employee {
         employee_id
      }
    }
    ... on Company {
      __typename
      legal_name
   }
}
```

```
{
  "data" : {
    "contacts": [
      {
        "__typename" : "Company",
        "display_name" : "Ada Lovelace Co.",
        "legal_name" :  "Lovelace Co.",
      },
      {
        "display_name" : "Sabrina Waserman",
        "birthday" : "1912-06-23"
        "employee_id" : "1",

      },
      {
        "display_name" : "Alan Turing",
        "birthday" : "1912-06-23"
      }
    ]
  }
}
```

Depending on whether or not `__typename` is present, we can infer the type on the client!

We can use this to our advantage!

```
query ContactsQuery($query: String!) {
  contacts(first: 1, query: $query) {
    display_name
    ... on Person {
      is_fulfilled_Person: __typename
      birthday
      ... on Employee {
        is_fulfilled_Employee : __typename
        employee_id
      }
    }
    ... on Company {
      is_fulfilled_Company : __typename
      legal_name
    }
  }
}
```

- We don't actually care about the `__typename` value, but we can use the presence of the aliases here to tell us what types resolved!

```
{
  "data" : {
    "contacts": [
      {
        "is_Company_fulfilled" : "Company",
        "display_name" : "Ada Lovelace Co.",
        "legal_name" :  "Lovelace Co.",
      },
      {
        "is_Person_fulfilled" : "Employee",
        "is_Employee_fulfilled" : "Employee",
        "display_name" : "Sabrina Wasserman",
        "birthday" : "1912-06-23"
        "employee_id" : "1",

      },
      {
        "is_Person_fulfilled" : "Customer",
        "is_Customer_fulfilled" : "Customer",
        "display_name" : "Alan Turing",
        "birthday" : "1912-06-23"
      }
    ]
  }
}
```


For every interface in the path, we receive type information! So we both know:

1. The concrete type of the object, and
2. All interfaces the object implements

Is_fulfilled Fields

We can add a field, `is_fulfilled<Type>`, to each type in the schema.

`is_fulfilled` has two key properties:

- 1) It's a boolean field
- 2) **It *always returns true***

The presence and absence of `is_fulfilled` is what determines if the type is fulfilled on the client!

Now, Consider Our Old Product Code!

```
if (model.isCompanyFulfilled) {
   displayCompany(model.displayName)
}
else if (model.isPersonFulfilled) {
   displayPerson(model.displayName)
}
```

Now, even if we have new interface type, or `Person` changes from a concrete to an interface type, `isPersonFulfilled` will still succeed.

```
Class SearchResult {

    Company getCompany() {
      if (self.isFulfilledCompany) {
        return reinterpretAsCompany(self);
      }
      return null;
    }

    Person getPerson() {
      if (self.isFulfilledPerson) {
        return reinterpretAsPerson(self);
      }
      return null;
    }
}
```

Now, field access doesn't break with new fulfilled field checks

Type check is also cheaper, as we're just checking a boolean!

@alias in Relay

```
function MyNode({nodeKey}) {
  const node = useFragment(graphql`
    fragment MyFragment on Node {
      ...MyViewer @alias(as: "my_viewer")
    }`, nodeKey);

  // Relay returns the fragment key as its own nullable property
  if(node.my_viewer == null) {
    return null;
  }

  // Because `my_viewer` is typed as nullable, Flow/TypeScript will
  // show an error if you try to use the `my_viewer` without first
  // performing a null check.
  //                          VVVVVVVVVVVVVV
  return <MyViewer viewerKey={node.my_viewer} />
}
```

- `@alias` in Relay gives similar functionality
- Essentially, it exposes a fragment as a named field within your selection

Bonus - `is_fulfilled` fields for @skip/@include

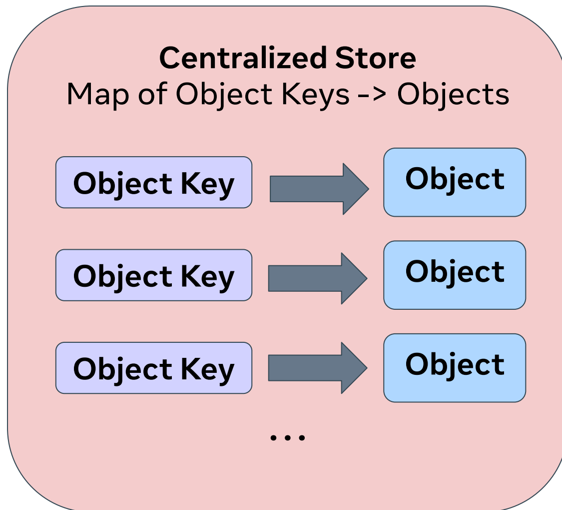
```
query Foo($skip:Boolean!) {
  test_root_field
    ... @skip(if: $skip) {
      is_skip_fulfilled: __typename
    }
    test_field
}
```

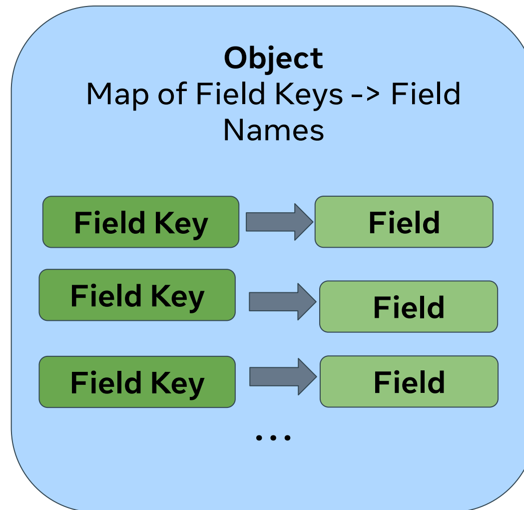
Because fulfilled fields just tell us whether or not a part of the query was executed, we can also use it to determine if portions of the query were skipped!

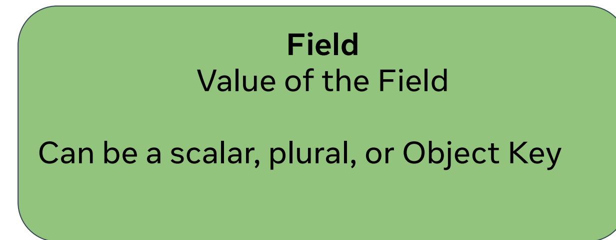
Why not just have `__fulfilled`, available on all types, and alias the usages?

Normalized Caching

- Normalized caching stores GraphQL objects in a simple key -> value pairing for easy caching
  - This is how [Relay's store cache]() works under the hood!
  - This is also core to [local data consistency with GraphQL]()







How do we select field keys?

```
query ContactsQueryA($query: String!) {
  contacts($query: String!) {
    myAlias: name
}

query ContactsQueryB($query: String!) {
  contacts($query: String!) {
    name
}
```

If we used aliases as part of the field keys, we would not consider these names to be the same!!

So, aliases **must be ignored** for normalized data in a consistency store

How do we select field keys?

```
query ContactsQuery($query: String!) {
  contacts($query: String!) {
    ... on Person {
       is_fulfilled_Person : __fulfilled
       ... on Employee {
       is_fulfilled_Employee : __fulfilled
      }
    }
    myAlias: __fulfilled
}
```

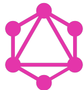
But, this would also mean we'd accidentally overwrite `__fulfilled` fields if we don't have typing information on the field name!

__typename and Normalized Caching

How do we decide what the `ObjectKey` is for each object?

- We want to use **some unique identifier for any non-ephemeral data**. That way, anytime we re-query the same object instance, the same object in the local store is updated, and any subscribers to that object are updated!

- We can depend on the **Global Object Identification specification** for this!
  - `id` and `__typename` can be used for a unique identifier
  - We can automatically inject these fields into queries before they're persisted to the server



GraphQL: https://graphql.org/learn/global-object-identification/

Problem 3 - `__typename` is expensive!

__typename can get expensive!

- `__typename` is a String field; querying for it on every object in a large query bloats payload size!
- We've found that querying for `__typename` on every object can lead to significant regressions

```
query ContactsQuery($query: String!) {
  a {
   b {
    c {
     d
    }
   }
  }
}
```

```
query ContactsQuery($query: String!) {
  __typename
  a {
   __typename
   b {
    __typename
    c {
      __typename
     d
    }
   }
  }
}
```

__typename can get expensive!

We can reduce this to just **the # of abstract types in the operation**, and store all the rest of the type information at compile time.

RFC: __fulfilled meta field #879

mjmahone commented on Jul 23, 2021 • edited

https://github.com/graphql/graphql-spec/pull/879

To summarize:

1. **Depending on `__typename` limits schema evolution for old clients.**
   - Where possible, use `is_fulfilled` fields instead to make clients more robust to schema changes.

2. `__typename` alone is not enough to determine if a given object implements an interface!

3. `__typename` **is expensive** to query for!
   - Where possible, use client-side metadata to supply typing for concrete types!

Thank You!

Sabrina Wasserman
Software Engineer