Understanding GraphQL Directives: Practical Use-Cases at Zalando

In this blog post, we dive into the practical applications of GraphQL directives at Zalando. With simple examples, we aim to highlight how they enhance our use cases. From defining precise authorization requirements to efficiently handling metadata, GraphQL directives offer flexibility and control in our API development process.

Boopathi Rajaa Nedunchezhiyan
Principal Engineer

Posted on Oct 19, 2023

Tags: GraphQL, APIs

GraphQL directives

In GraphQL, if you've used the syntax that starts with an `@`, for example, `@foo`, then you've used GraphQL directives. Directives provide a way to extend the language features of GraphQL using a supported syntax. Certain directives are built into GraphQL, like `@skip`, `@include`, `@deprecated`, and `@specifiedBy`, and are supported by all GraphQL engines.

If we look closer, we can see that two of these directives (`@skip` and `@include`) are used only in the queries, and the other two (`@deprecated` and `@specifiedBy`) are used only in the schema. This is because GraphQL directives are defined for two different categories of locations - `TypeSystem` and `ExecutableDefinition`. The `TypeSystem` directives are defined for the schema, and the `ExecutableDefinition` directives are defined for the queries. We will discuss this in detail in the next section.

The query directives are generally useful for clients to express certain types of metadata for the query. The schema directives are generally useful for declaratively specifying common server-side behaviors, for example, authorization requirements, marking sensitive data, etc.

Part 1: Schema directives at Zalando

The schema directives refer to the directives defined for the `TypeSystem` locations. The type system directives are available for the locations listed below. Consider `@foo` a directive for the location mentioned in the 1st column.

```
directive @foo on LOCATION_IN_FIRST_COLUMN
```

Directive Location	Example
SCHEMA	`schema @foo { query: Query }`
SCALAR	`scalar x @foo`
OBJECT	`type Product @foo { }`
FIELD_DEFINITION	`type X { field: String @foo }`
ARGUMENT_DEFINITION	`type X { field(arg: Int @foo): String }`
INTERFACE	`interface X @foo {}`
UNION	`union X @foo = A | B`
ENUM	`enum X @foo { A B }`
ENUM_VALUE	`enum X { A @foo B }`
INPUT_OBJECT	`input X @foo { }`
INPUT_FIELD_DEFINITION	`input X { field: String @foo }`

The guild - https://the-guild.dev has a great article and a mechanism for implementing schema directives via their graphql-tools packages. I highly recommend reading it and using graphql-tools for implementing schema directives.

The gist is that you can define a directive in the schema and implement the directive in the resolver layer. The directive is implemented as a function that takes the resolver function as an argument and returns a new resolver function. The new resolver function can be used to implement the directive logic.

You can think of schema directives as some function call injected to your resolver function in a declarative way. Consider the following illustration to understand where the directive

function can be invoked in the context of a resolver.

```
/**
 * Illustration of schema directives execution in
 * the query execution pipeline
 */
const resolvers = {
  Query: {
    async product(_, { id }) {
      // schema directives
      schemaDirectivesExecutions();

      // resolver logic
      const product = await getProduct(id);

      // schema directives
      schemaDirectivesExecutions();

      return product;
    },
  },
};
```

`@isAuthenticated`

At Zalando, we use SSO for customer authentication and step-up authentication. Our GraphQL server handles publicly available data like the product data, and also handles confidential data like customer-related data.

The queries can contain customer fields along with product fields and other non-customer data. Here, we need to ensure that the customer is authenticated and has the correct authenticity levels (ACR Value) whenever a field or mutation containing customer information is used in the query. So, we need a way to control this granularly for different data points in the schema. The directive `@isAuthenticated` is used for this purpose.

The directive is defined in the schema as follows -

```
scalar ACRValue @specifiedBy(url: "https://example.com/zalando-acr-value")

directive @isAuthenticated(
  """
  The ACR value, which indicates the level of authenticity
  expected to perform the operation.

  Optional. If not provided, the default behavior is to simply
  validate a user is authenticated and has no ACR requirements.
  """
  acrValue: ACRValue
) on FIELD_DEFINITION
```

For example, it is used in a mutation definition as follows -

```
type Query {
  customer: Customer @isAuthenticated
}
type Mutation {
  updateCustomerInfo(
    email: String
    phoneNumber: String
  ): UpdateCustomerInfoResult @isAuthenticated(acrValue: HIGH)
}
```

@sensitive

We expose customer-sensitive data via our GraphQL API - like the email address, customer name, phone number, address, etc, to render the customer profile page. We also use observability tools and monitoring tools like logging and tracing. We do not want such sensitive customer data in the logs and traces. So, we need a way to control logging so that the logs contain enough information to debug issues but not sensitive customer data. The directive `@sensitive` is used for this purpose.

```
directive @sensitive(
  "An optional reason why the field is marked as sensitive"
  reason: String
) on ARGUMENT_DEFINITION
```

For example, it is used in a mutation definition as follows -

```
type Mutation {
  updateCustomerInfo(
    email: String @sensitive(reason: "Customer email address")
    phoneNumber: String @sensitive(reason: "Customer phone number")
  ): UpdateCustomerInfoResult
}
```

It could be somewhat manual and forgetful to add `@sensitive` to the correct arguments in the schema proactively. So, we also rely on a schema linter to automatically fail when a field/argument name contains sensitive keywords like `password`, `email`, `phone`, `bank`, `bic`, `account`, `owner`, `order`, `token`, `voucher`, `customer`, etc. This way, we can ensure we do not forget to add `@sensitive` to the correct fields/arguments.

Implementing this directive is also quite simple and does not require any resolver logic. It can be implemented in NodeJS as follows (the implementation is shortened to fit into a post) -

```
function getSensitiveVariables(schema, document) {
  const sensitiveVariables = [];
  require("graphql").validate(schema, document, [
    (context) => ({
      Variable(node) {
        const isSensitive = context
          .getArgument()
```

```
            ?.astNode?.directives?.some(
              (directive) => directive.name.value === "sensitive"
            );
          if (isSensitive) {
            sensitiveVariables.push(node.name.value);
          }
        },
      }),
    ]);
    return sensitiveVariables;
  }
```

`@requireExplicitEndpoint`

With GraphQL, all of the varieties of HTTP requests fit into one single pattern - `POST /graphql` . It makes using techniques and tools available for REST APIs - like rate limiting, bot protection, caching, and other security practices fail to work out of the box. So, we need a way to control different schema sections to be exposed via different HTTP endpoints. The directive `@requireExplicitEndpoint` is used for this purpose.

```
directive @requireExplicitEndpoint(endpoints: [String!]!) on FIELD_DEFINIT
```

In implementing this directive, we override the resolver for the respective field where it is used. We can access the request parameters (like pathname) by running GraphQL over HTTP. We then match the pathname with the list of endpoints provided in the directive and return an error if there is no match.

This directive allows us to define custom routes for different schema sections and prevents the client from accessing the entire schema via a single HTTP endpoint, `POST /graphql`. For example, let's see how we can define this directive for the `updateDeliveryAddress` mutation.

```
type Mutation {
  updateDeliveryAddress(
    id: ID!
    newAddress: CustomerAddress!
  ): UpdateDeliveryAddressResult
    @requireExplicitEndpoint(endpoints: ["/customer-addresses"])
}
```

So, a mutation query like the following will fail with an error when executing via `/graphql` endpoint -

```
# POST /graphql
mutation {
  updateDeliveryAddress(id: "1234", newAddress: { name: "Boopathi" }) {
    id
```

```
    }
}
```

`@draft , @allowedFor`

We use persisted queries and define different schema stability levels for different sections of the schema. We have a separate blog post explaining the details of how Zalando uses persisted queries and how we think about schema stability and granular control.

The `@draft` and `@allowedFor` directives are used for this purpose. It prevents clients from persisting a query that is not stable yet.

```
# Draft
directive @draft on FIELD_DEFINITION

# Restricted usage: Only for the specified components
directive @component(name: String!) on QUERY
directive @allowedFor(componentNames: [String!]!) on FIELD_DEFINITION
```

`@final`

Enums in GraphQL are tricky to evolve. Adding a new value to an enum is not considered a breaking change, but it is still a "dangerous" change. It is "dangerous" because the client might not have a handler for the new value. It is easy to update the client code for web applications, but for the mobile native apps shipped to the app store, it is impossible to update the client code. Though we practice defensive coding practices to handle unknown values, we still need a way to control the evolution of enums in a safe manner. The directive `@final` is used for this purpose.

```
directive @final on ENUM
```

The implementation of this directive is absolutely nothing - i.e., it does not need any runtime behavior. It is only used in our GraphQL linter that executes during the build time and prevents additions of new values to enums which are marked as final. When we want to make a dangerous change, we remove the `@final` directive in the first pull request and reason about and find if old apps would break by making this "dangerous" change. After extending the enum, we add it in a separate pull request. This process is cumbersome, but it is on purpose. It must be more complicated to make dangerous changes, and it is a trade-off we are willing to make.

The ideal situation would be that all enums are treated as final by default, and this directive is never required in the first place. During schema evolution, your use case might warrant such

directives to control a smooth schema evolution.

`@extensibleEnum`

As we are discussing enums, another use-case of directives for enums, primarily one-off use cases, and extending them is the common case. Creating enums for one use case is tricky in these cases, and extending it has dangerous consequences. At Zalando, we have RESTful API guidelines, and one of the recommendations is to use x-extensible-enum to represent all enums. This recommendation is so that the enums can evolve, and the client is aware, right from the name, that it is extensible. We use the directive `@extensibleEnum` for this purpose. The type in GraphQL for the field would be `String`, and the directive is used to provide the list of allowed values.

```
directive @extensibleEnum(values: [String!]!) on FIELD_DEFINITION
```

For example, it is used in a query definition as follows -

```
type CustomerConsent {
  status: String! @extensibleEnum(values: ["GRANTED", "REJECTED"])
}
```

With `@extensibleEnum`, we found that contributors to the schema are more likely to think about the evolution of schema. We also noticed that contributors are more likely to use this directive for defining enums than the GraphQL native enum, as this directive is more explicit about the extensibility of the enum.

`@resolveEntityId`

Our GraphQL schema defines certain types as Entities related to the Entity-Relationship model. We define entities abstractly as the basic building blocks for designing customer experience. For example, product, customer, brand, etc. are some entities. The entity definition has some properties -

- it follows a specific template/pattern of resolvers that is mostly the same for all entities
- it is of a specific type name as defined in the schema
- it has a unique ID of a specific pattern (for example, `entity:product:1234` for `type Product`)
- it has a set of fields that are common to all entities

To solve these cases holistically, we use the directive `@resolveEntityId` defined against each entity definition in the schema.

```
directive @resolveEntityId(
  "An optional override name for the entity name in its ID"
  override: String
) on OBJECT
```

The usage is as follows -

```
type Product implements Entity @resolveEntityId {
  id: ID!
}
```

The implementation of this directive is two-fold. For one, we generate TypeScript code based on the `resolveEntityId` directive. This code generation allows us to develop the boilerplate code for the entity ID type definitions and resolvers - for example, the `__typename` resolvers. The other part is the runtime, where an `id` resolver is added to wrap the entity IDs - for example, consider the product - `entity:product:1234` is the full entity ID, and the `1234` is called the SKU of the product.

Part 2: Query directives at Zalando

Query directives are directives that are defined for the `ExecutableDefinition` locations. The executable directives are available for the locations listed below. Consider `@foo` a directive for the location mentioned in the 1st column.

```
directive @foo on LOCATION_IN_FIRST_COLUMN
```

Directive Location	Example
QUERY	`query name @foo {}`
MUTATION	`mutation name @foo {}`
SUBSCRIPTION	`subscription name @foo {}`
FIELD	`query { product @foo {} }`
FRAGMENT_DEFINITION	`fragment x on Query @foo { }`

Directive Location	Example
FRAGMENT_SPREAD	`query { ...x @foo }`
INLINE_FRAGMENT	`query { ... @foo { } }`
VARIABLE_DEFINITION	`query ($id: ID @foo) { }`

Unlike schema directives, graphql-tools does not support attaching functions to resolvers the same way for query directives. They also have an excellent point: query directives are good for annotating the query with metadata and not for resolver logic. Likewise, most of our use cases include attaching metadata at the query level and one case for observability and monitoring.

For query metadata, the implementation is as simple as going through the parsed GraphQL document (AST - Abstract Syntax Tree) and extracting the metadata from the query directives. We use a two-step approach for the use case that adds behavior to a field - specifically the `@omitErrorTag` directive (discussed below). In the first step before execution, we extract the field paths of the fields that have this directive. In the second step, after execution, we match the error paths and omit the error tag for those extracted paths.

`@component`

The `@component` directive defines a component name by the client for the query. This directive is used in our observability and monitoring tools and for schema stability - restricted usage in production. See our blog post GraphQL persisted queries and schema stability for more details.

```
directive @component(name: String!) on QUERY
```

`@tracingTag`

The `@tracingTag` directive defines an OpenTelemetry tracing tag for the query. Using this directive on a query adds a specific client-defined tag to our tracing spans. The clients can then follow the traces and filter by this tag to find the traces for a particular query. This directive is useful for debugging, troubleshooting, monitoring specific set of queries, etc.

```
directive @tracingTag(value: String!) on QUERY | MUTATION | SUBSCRIPTION
```

@omitErrorTag

The `@omitErrorTag` directive is used to omit marking the tracing span as an error. This directive can be used on a particular field in the query. This directive lets the client define that some field errors are noncritical and should not be reported for alerting. The 24x7 on-call team can then focus on the critical errors and not be distracted by the noise.

```
directive @omitErrorTag on FIELD
```

@maxCountInBatch

The `@maxCountInBatch` directive is used at the Query level to declare the maximum number of queries that can be batched together in a single request. This directive is client-controlled i.e. it is only available during build/persist time. At runtime, the directive is used to prevent overfetching of data and bot abuse of the GraphQL API.

Our GraphQL server allows batching of multiple queries in a single batch. With persisted queries, we only send the id of the query, and the client cannot send a raw query in production. So, the system design allows the safe usage of `maxCountInBatch` controlled by the clients.

```
directive @maxCountInBatch(value: Int!) on QUERY
```

Example usage of all of the above query directives

```
query product_card($id: ID!)
#
# component directive
@component(name: "web-product-card")
#
# tracing tag directive to add a tag to the tracing span
@tracingTag(value: "slo-1s")
#
# maxCountInBatch directive to limit the number of queries in a batch requ
@maxCountInBatch(value: 50) {
  product(id: $id) {
    id
    name
    brand {
      id
      name
    }
    # omitErrorTag directive to omit marking the tracing
    # span as an error if inWishlist field errors
    inWishlist @omitErrorTag
  }
}
```

Conclusion

Query directives allow clients to define metadata and, on rare occasions, behavior. Schema directives, on the other hand, allow the server to define behavior, validation, and resolution logic in a declarative manner. Schema directives carry the added advantage that the servers can make breaking changes to these directives, as these directives are not consumed by the client - they only experience the resulting behavior. It's important when designing a directive to consider its properties, use cases, trade-offs, and where the control should lie.

The use cases outlined in this blog post represent some of the ways we use GraphQL directives at Zalando. There are numerous other cases that we'll cover in future blog posts. I hope this piece provides a good starting point for you to explore GraphQL directives and their practical applications.

If you would like to work on similar challenges, consider joining our engineering teams.

Further reading

- Schema Directives - GraphQL Tools
- GraphQL persisted queries and Schema stability
- Modeling Errors in GraphQL
- Optimize GraphQL Server with Lookaheads

Related posts

GraphQL persisted queries and Schema stability

Learn how Zalando uses persisted queries, and how we define and think about different levels of stability of our... Read more...

Boopathi Rajaa Nedunchezhiyan
Senior Software Engineer

Feb 17
2022

Modeling Errors in GraphQL

GraphQL excels in modeling data requirements. Modeling errors as schema types in GraphQL is required for certain... Read more...

Boopathi Rajaa Nedunchezhiyan
Senior Software Engineer

Apr 13
2021

Optimize GraphQL Server with Lookaheads

GraphQL offers a way to optimize the data between a client and a server. We can use the declarative nature of a... Read more...

Boopathi Rajaa Nedunchezhiyan
Senior Software Engineer

Mar 18
2021

Follow us