GraphQL Query Options

Intro

This page will walk you through a series of GraphQL queries, each designed to demonstrate a particular feature of GraphQL. These examples will work on the *real* schema used on graphql-reference example. You can run this example locally to experiment and poke around the innards of the site! To get to the GraphiQL editor, go to `localhost:8000/___graphql` (that's three underscores). You can also open the CodeSandbox version of the example site.

For more background information, read about why Gatsby uses GraphQL and how to use GraphiQL in any Gatsby site.

Basic queries

Example: Site metadata

Start with the basics, pulling up the site `title` from your `gatsby-config.js`'s `siteMetadata`.

```
{
  site {
    siteMetadata {
      title
    }
  }
}
```

In the GraphiQL editor, try editing the query to include the `description` from `siteMetadata`. When typing in the query editor you can use `Ctrl + Space` to see autocomplete options and `Ctrl + Enter` to run the current query.

Example: Multiple data nodes

Gatsby structures its content as collections of `nodes`, which are connected to each other with `edges`. In this query you ask for the total count of plugins in this Gatsby site, along with specific information about each one.

```
{
  allSitePlugin {
```

```
    totalCount
    edges {
      node {
        name
        version
        packageJson {
          description
        }
      }
    }
  }
}
```

In the GraphiQL editor, try using the editor's autocomplete (`Ctrl + Space`) to get extended details from the `packageJson` nodes.

If you're using Gatsby version `2.2.0` or later, you can remove `edges` and `node` from your query and replace it with `nodes`. The query will still work and the returned object will reflect the `nodes` structure.

```
{
  allSitePlugin {
    totalCount
    nodes {
      name
      version
      packageJson {
        description
      }
    }
  }
}
```

Field arguments

This section covers the different arguments you can pass in to GraphQL fields.

Limit

There are several ways to reduce the number of results from a query. Here `totalCount` tells you there's 8 results, but `limit` is used to show only the first three.

```
{
  allMarkdownRemark(limit: 3) {
    totalCount
    edges {
      node {
```

```
        frontmatter {
          title
        }
      }
    }
  }
}
```

Skip

Skip over a number of results. In this query `skip` is used to omit the first 3 results.

```
{
  allMarkdownRemark(skip: 3) {
    totalCount
    edges {
      node {
        frontmatter {
          title
        }
      }
    }
  }
}
```

Filter

In this query, the `filter` argument and the `ne` (not equals) operator are used to show only results that have a title. You can find a good video tutorial on this in LevelUpTuts Gatsby Tutorial #9: Filters & Sorting with GraphQL.

```
{
  allMarkdownRemark(filter: { frontmatter: { title: { ne: "" } } }) {
    totalCount
    edges {
      node {
        frontmatter {
          title
        }
      }
    }
  }
}
```

Complete list of possible operators Gatsby relies on Sift to enable MongoDB-like query syntax for object filtering. This allows Gatsby to support

operators like `eq`, `ne`, `in`, `regex` and querying nested fields through the `___` connector.

In the code block below the list, there is an example query with a description of what the query does for each operator.

- `eq`: short for **equal**, must match the given data exactly
- `ne`: short for **not equal**, must be different from the given data
- `regex`: short for **regular expression**, must match the given pattern
- `glob`: short for **global**, allows to use wildcard `*` which acts as a placeholder for any non-empty string
- `in`: short for **in array**, must be an element of the array
- `nin`: short for **not in array**, must NOT be an element of the array
- `gt`: short for **greater than**, must be greater than given value
- `gte`: short for **greater than or equal**, must be greater than or equal to given value
- `lt`: short for **less than**, must be less than given value
- `lte`: short for **less than or equal**, must be less than or equal to given value
- `elemMatch`: short for **element match**, this indicates that the field you are filtering will return an array of elements, on which you can apply a filter using the previous operators

```
{
  # eq: I want all the titles that match "Fantastic Beasts and Where to Find Them"
  example_eq: allMarkdownRemark(
    filter: {
      frontmatter: { title: { eq: "Fantastic Beasts and Where to Find Them" } }
    }
  ) {
    edges {
      node {
        frontmatter {
          title
        }
      }
    }
  }

  # neq: I want all the titles which are NOT equal to the empty string
  example_ne: allMarkdownRemark(
    filter: { frontmatter: { title: { ne: "" } } }
  ) {
    edges {
      node {
        frontmatter {
          title
```

```
      }
    }
  }
}

# regex: I want all the titles that do not start with 'T' -- this is what /^[^T]/ means.
# To learn more about regular expressions: https://regexr.com/
example_regex: allMarkdownRemark(
  filter: { frontmatter: { title: { regex: "/^[^T]/" } } }
) {
  edges {
    node {
      frontmatter {
        title
      }
    }
  }
}

# glob: I want all the titles that contain the word 'History'.
# The wildcard * stands for any non-empty string.
example_glob: allMarkdownRemark(
  filter: { frontmatter: { title: { glob: "*History*" } } }
) {
  edges {
    node {
      frontmatter {
        title
      }
    }
  }
}

# in: I want all the titles and dates from `frontmatter`
# where the title is either
# - "Children's Anthology of Monsters", or
# - "Hogwarts: A History".
example_in: allMarkdownRemark(
  filter: {
    frontmatter: {
      title: {
        in: ["Children's Anthology of Monsters", "Hogwarts: A History"]
      }
    }
  }
) {
```

```
  edges {
    node {
      frontmatter {
        title
        date
      }
    }
  }
}

# nin: I want all the titles and dates from `frontmatter`
# where the title is neither
# - "Children's Anthology of Monsters", nor
# - "Hogwarts: A History".
example_nin: allMarkdownRemark(
  filter: {
    frontmatter: {
      title: {
        nin: ["Children's Anthology of Monsters", "Hogwarts: A History"]
      }
    }
  }
) {
  edges {
    node {
      frontmatter {
        title
        date
      }
    }
  }
}

# lte: I want all the titles for which `timeToRead` is less than or equal to 4 minutes.
example_lte: allMarkdownRemark(filter: { timeToRead: { lte: 4 } }) {
  edges {
    node {
      frontmatter {
        title
      }
    }
  }
}

# elemMatch: I want to know all the plugins that contain "chokidar" in their dependencies
# Note: the `allSitePlugin` query lists all the plugins used in our Gatsby site.
```

```
  example_elemMatch: allSitePlugin(
    filter: {
      packageJson: { dependencies: { elemMatch: { name: { eq: "chokidar" } } } }
    }
  ) {
    edges {
      node {
        name
      }
    }
  }
}
```

If you want to understand more how these filters work, looking at the corresponding tests in the codebase could be very useful.

Filtering on multiple fields It is also possible to filter on multiple fields by separating the individual filters by a comma (which works as an AND):

```
filter: { contentType: { in: ["post", "page"] }, draft: { eq: false } }
```

In this query the fields `categories` and `title` are filtered to find the book that belongs to the `magical creatures` category *AND* has `Fantastic` in its title.

```
{
  allMarkdownRemark(
    filter: {
      frontmatter: {
        categories: { in: ["magical creatures"] }
        title: { regex: "/Fantastic/" }
      }
    }
  ) {
    totalCount
    edges {
      node {
        frontmatter {
          title
        }
      }
    }
  }
}
```

Combining operators You can also combine the mentioned operators. This query filters on `/History/` for the `regex` operator, which would return `Hogwarts: A History` and `History of Magic`. Then the `ne` operator filters

out `History of Magic`, so the final result contains only `Hogwarts: A History`.

```
{
  allMarkdownRemark(
    filter: {
      frontmatter: { title: { regex: "/History/", ne: "History of Magic" } }
    }
  ) {
    totalCount
    edges {
      node {
        frontmatter {
          title
        }
      }
    }
  }
}
```

Sort

The ordering of your results can be specified with `sort`. Here the results are sorted in ascending order of `frontmatter`'s `date` field.

```
{
  allMarkdownRemark(sort: { fields: [frontmatter___date], order: ASC }) {
    totalCount
    edges {
      node {
        frontmatter {
          title
          date
        }
      }
    }
  }
}
```

Sorting on multiple fields You can also sort on multiple fields but the `sort` keyword can only be used once. The second sort field gets evaluated when the first field (here: `date`) is identical. The results of the following query are sorted in ascending order of `date` and `title` field.

```
{
  allMarkdownRemark(
    sort: { fields: [frontmatter___date, frontmatter___title], order: ASC }
  ) {
```

```
    totalCount
    edges {
      node {
        frontmatter {
          title
          date
        }
      }
    }
  }
}
```

`Children's Anthology of Monsters` and `Break with Banshee` both have the same date (`1992-01-02`) but in the first query (only one sort field) the latter comes after the first. The additional sorting on the `title` puts `Break with Banshee` in the right order.

Sort order By default, sort `fields` will be sorted in ascending order. Optionally, you can specify a sort `order` per field by providing an array of `ASC` (for ascending) or `DESC` (for descending) values. For example, to sort by `frontmatter.date` in ascending order, and additionally by `frontmatter.title` in descending order, you would use `sort: { fields: [frontmatter___date, frontmatter___title], order: [ASC, DESC] }`. Note that if you only provide a single sort `order` value, this will affect the first sort field only, the rest will be sorted in default ascending order.

```
{
  allMarkdownRemark(
    sort: {
      fields: [frontmatter___date, frontmatter___title]
      order: [ASC, DESC]
    }
  ) {
    totalCount
    edges {
      node {
        frontmatter {
          title
          date
        }
      }
    }
  }
}
```

Formatting

Dates Dates can be formatted using the `formatString` function.

```
{
  allMarkdownRemark(filter: { frontmatter: { date: { ne: null } } }) {
    edges {
      node {
        frontmatter {
          title
          date(formatString: "dddd DD MMMM YYYY")
        }
      }
    }
  }
}
```

Gatsby relies on Moment.js to format the dates. This allows you to use any tokens in your string. See the Moment.js documentation for more tokens.

You can also pass in a `locale` to adapt the output to your language. The above query gives you the English output for the weekdays, this example outputs them in German.

```
{
  allMarkdownRemark(filter: { frontmatter: { date: { ne: null } } }) {
    edges {
      node {
        frontmatter {
          title
          date(formatString: "dddd DD MMMM YYYY", locale: "de-DE")
        }
      }
    }
  }
}
```

Example: `anotherDate(formatString: "dddd, MMMM Do YYYY, h:mm:ss a") # Sunday, August 5th 2018, 10:56:14 am`

Dates also accept the `fromNow` and `difference` function. The former returns a string generated with Moment.js' `fromNow` function, the latter returns the difference between the date and current time (using Moment.js' `difference` function).

```
{
  one: allMarkdownRemark(
    filter: { frontmatter: { date: { ne: null } } }
    limit: 2
```

```
  ) {
    edges {
      node {
        frontmatter {
          title
          date(fromNow: true)
        }
      }
    }
  }
  two: allMarkdownRemark(
    filter: { frontmatter: { date: { ne: null } } }
    limit: 2
  ) {
    edges {
      node {
        frontmatter {
          title
          date(difference: "days")
        }
      }
    }
  }
}
```

Excerpt Excerpts accept three options: `pruneLength`, `truncate`, and `format`. `format` can be `PLAIN` or `HTML`.

```
{
  allMarkdownRemark(filter: { frontmatter: { date: { ne: null } } }, limit: 5) {
    edges {
      node {
        frontmatter {
          title
        }
        excerpt(format: PLAIN, pruneLength: 200, truncate: true)
      }
    }
  }
}
```

Example: Sort, filter, limit & format together

This query combines sorting, filtering, limiting and formatting together.

```
{
  allMarkdownRemark(
```

```
    limit: 3
    filter: { frontmatter: { date: { ne: null } } }
    sort: { fields: [frontmatter___date], order: DESC }
  ) {
    edges {
      node {
        frontmatter {
          title
          date(formatString: "dddd DD MMMM YYYY")
        }
      }
    }
  }
}
```

Query variables

In addition to adding query arguments directly to query fields, GraphQL allows you to pass in "query variables". These can be both simple scalar values as well as objects.

The query below is the same one as the previous example, but with the input arguments passed in as "query variables".

To add variables to page component queries, pass these in the `context` object when creating pages.

```
query GetBlogPosts(
  $limit: Int
  $filter: MarkdownRemarkFilterInput
  $sort: MarkdownRemarkSortInput
) {
  allMarkdownRemark(limit: $limit, filter: $filter, sort: $sort) {
    edges {
      node {
        frontmatter {
          title
          date(formatString: "dddd DD MMMM YYYY")
        }
      }
    }
  }
}

# Query Variables
{
  "limit": 5,
  "filter": {
```

```
    "frontmatter": {
      "date": {
        "ne": null
      }
    }
  },
  "sort": {
    "fields": "frontmatter___title",
    "order": "DESC"
  }
}
```

Group

You can also group values on the basis of a field (like the title, date, or category) and get the field value, the total number of occurrences, and the edges.

The query below gets you all categories (`fieldValue`) applied to a book and the number of books (`totalCount`) a given category is applied to. In addition, you are grabbing the `title` of books in a given category. For example, the response for this query contains 3 books in the `magical creatures` category.

```
{
  allMarkdownRemark(filter: { frontmatter: { title: { ne: "" } } }) {
    group(field: frontmatter___categories) {
      fieldValue
      totalCount
      edges {
        node {
          frontmatter {
            title
          }
        }
      }
    }
    nodes {
      frontmatter {
        title
        categories
      }
    }
  }
}
```

Fragments

Fragments are a way to save frequently used queries for reuse.

To create a fragment, define it in a query and export it as a named export from any file Gatsby is aware of. A fragment is available for use in any other GraphQL query, regardless of its location in the project.

Fragments are globally defined in a Gatsby project, so names have to be unique.

The query below defines a fragment to get the site title, and then uses the fragment to access this information.

```
fragment fragmentName on Site {
  siteMetadata {
    title
  }
}

{
  site {
    ...fragmentName
  }
}
```

Aliasing

Want to run two queries on the same datasource? You can do this by aliasing your queries. See the query below for an example:

```
{
  someEntries: allMarkdownRemark(skip: 3, limit: 3) {
    edges {
      node {
        frontmatter {
          title
        }
      }
    }
  }
  someMoreEntries: allMarkdownRemark(limit: 3) {
    edges {
      node {
        frontmatter {
          title
        }
      }
    }
  }
}
```

When you use your data, you will be able to reference it using the alias instead

of the root query name. In this example, that would be `data.someEntries` or `data.someMoreEntries` (instead of `data.allMarkdownRemark`).

The same works for fields inside a query. Take this example:

```
{
  allMarkdownRemark(skip: 3, limit: 3) {
    edges {
      node {
        frontmatter {
          header: title
          date
          relativeDate: date(fromNow: true)
        }
      }
    }
  }
}
```

Instead of receiving `title` you'll get `header`. This is especially useful when you want to display the same field in different ways as the `date` shows. You both get `date` and `relativeDate` from the same source.

Conditionals

GraphQL allows you to skip a piece of a query depending on variables. This is handy when you need to render some part of a page conditionally.

In the GraphiQL editor, try changing variable `withDate` in the example query below:

```
query GetBlogPosts($withDate: Boolean = false) {
  allMarkdownRemark(limit: 3, skip: 1) {
    edges {
      node {
        frontmatter {
          title
          date @include(if: $withDate)
        }
      }
    }
  }
}
```

Use directive `@include(if: $variable)` to conditionally include a part of a query or `@skip(if: $variable)` to exclude it.

You can use those directives on any level of the query and even on fragments. Take a look at an advanced example:

```
query GetBlogPosts($preview: Boolean = true) {
  allMarkdownRemark(limit: 3, skip: 1) {
    edges {
      node {
        ...BlogPost @skip(if: $preview)
        ...BlogPostPreview @include(if: $preview)
      }
    }
  }
  allFile(limit: 2) @skip(if: $preview) {
    edges {
      node {
        relativePath
      }
    }
  }
}

fragment BlogPost on MarkdownRemark {
  html
  frontmatter {
    title
    date
  }
}

fragment BlogPostPreview on MarkdownRemark {
  excerpt
  frontmatter {
    title
  }
}
```