Recipes: Querying Data

Gatsby lets you access your data across all sources using a single GraphQL interface.

Querying data with a Page Query

You can use the `graphql` tag to query data in the pages of your Gatsby site. This gives you access to anything included in Gatsby's data layer, such as site metadata, source plugins, images, and more.

Directions

1. Import `graphql` from `gatsby`.
2. Export a constant named `query` and set its value to be a `graphql` template with the query between two backticks.
3. Pass in `data` as a prop to the component.
4. The `data` variable holds the queried data and can be referenced in JSX to output HTML.

```
import React from "react"
// highlight-next-line
import { graphql } from "gatsby"

import Layout from "../components/layout"

// highlight-start
export const query = graphql`
  query HomePageQuery {
    site {
      siteMetadata {
        title
      }
    }
  }
`
// highlight-end
// highlight-next-line
const IndexPage = ({ data }) => (
  <Layout>
    // highlight-next-line
    <h1>{data.site.siteMetadata.title}</h1>
  </Layout>
)

export default IndexPage
```

Additional resources

- GraphQL and Gatsby: understanding the expected shape of your data
- More on querying data in pages with GraphQL
- MDN on Tagged Template Literals like the ones used in GraphQL

Querying data with the StaticQuery Component

`StaticQuery` is a component for retrieving data from Gatsby's data layer in non-page components, such as a header, navigation, or any other child component.

Directions

1. The `StaticQuery` Component requires two render props: `query` and `render`.

```
import React from "react"
import { StaticQuery, graphql } from "gatsby"

const NonPageComponent = () => (
  <StaticQuery
    query={graphql` // highlight-line
      query NonPageQuery {
        site {
          siteMetadata {
            title
          }
        }
      }
    `}
    render={(
      data // highlight-line
    ) => (
      <h1>
        Querying title from NonPageComponent with StaticQuery:
        {data.site.siteMetadata.title}
      </h1>
    )}
  />
)

export default NonPageComponent
```

2. You can now use this component as you would any other component by importing it into a larger page of JSX components and HTML markup.

Querying data with the useStaticQuery hook

Since Gatsby v2.1.0, you can use the `useStaticQuery` hook to query data with a JavaScript function instead of a component. The syntax removes the need for a `<StaticQuery>` component to wrap everything, which some people find simpler to write.

The `useStaticQuery` hook takes a GraphQL query and returns the requested data. It can be stored in a variable and used later in your JSX templates.

Prerequisites

- You'll need React and ReactDOM 16.8.0 or later (keeping Gatsby updated handles this)
- Recommended reading: the Rules of React Hooks

Directions

1. Import `useStaticQuery` and `graphql` from `gatsby` in order to use the hook query the data.
2. In the start of a stateless functional component, assign a variable to the value of `useStaticQuery` with your `graphql` query passed as an argument.
3. In the JSX code returned from your component, you can reference that variable to handle the returned data.

```
import React from "react"
import { useStaticQuery, graphql } from "gatsby" //highlight-line

const NonPageComponent = () => {
  // highlight-start
  const data = useStaticQuery(graphql`
    query NonPageQuery {
      site {
        siteMetadata {
          title
        }
      }
    }
  `)
  // highlight-end
  return (
    <h1>
      Querying title from NonPageComponent: {data.site.siteMetadata.title}{" "}
      //highlight-line
    </h1>
  )
}

export default NonPageComponent
```

Additional resources

- More on Static Query for querying data in components
- The difference between a static query and a page query
- More on the useStaticQuery hook
- Visualize your data with GraphiQL

Limiting with GraphQL

When querying for data with GraphQL, you can limit the number of results returned with a specific number. This is helpful if you only need a few pieces of data or need to paginate data.

To limit data, you'll need a Gatsby site with some nodes in the GraphQL data layer. All sites have some nodes like `allSitePage` and `sitePage` created automatically: more can be added by installing source plugins like `gatsby-source-filesystem` in `gatsby-config.js`.

Prerequisites

- A Gatsby site

Directions

1. Run `gatsby develop` to start the development server.
2. Open a tab in your browser at: `http://localhost:8000/___graphql`.
3. Add a query in the editor with the following fields on `allSitePage` to start off:

```
{
  allSitePage {
    edges {
      node {
        id
        path
      }
    }
  }
}
```

4. Add a `limit` argument to the `allSitePage` field and give it an integer value `3`.

```
{
  allSitePage(limit: 3) { // highlight-line
    edges {
      node {
        id
        path
      }
    }
  }
}
```

5. Click the play button in the GraphiQL page and the data in the `edges` field will be limited to the number specified.

Additional resources

- Learn about nodes in Gatsby's GraphQL data API
- Gatsby GraphQL reference for limiting
- Live example:

Sorting with GraphQL

The ordering of your results can be specified with the GraphQL `sort` argument. You can specify which fields to sort by and the order to sort in.

For this recipe, you'll need a Gatsby site with a collection of nodes to sort in the GraphQL data layer. All sites have some nodes like `allSitePage` created automatically: more can be added by installing source plugins.

Prerequisites

- A Gatsby site
- Queryable fields prefixed with `all`, e.g. `allSitePage`

Directions

1. Run `gatsby develop` to start the development server.
2. Open the GraphiQL explorer in a browser tab at: `http://localhost:8000/___graphql`
3. Add a query in the editor with the following fields on `allSitePage` to start off:

```
{
  allSitePage {
    edges {
      node {
        id
        path
      }
    }
  }
}
```

4. Add a `sort` argument to the `allSitePage` field and give it an object with the `fields` and `order` attributes. The value for `fields` can be a field or an array of fields to sort by (this example uses the `path` field), the `order` can be either `ASC` or `DESC` for ascending and descending respectively.

```
{
  allSitePage(sort: {fields: path, order: ASC}) { // highlight-line
    edges {
      node {
        id
        path
      }
    }
  }
}
```

5. Click the play button in the GraphiQL page and the data returned will be sorted ascending by the `path` field.

Additional resources

- Gatsby GraphQL reference for sorting
- Learn about nodes in Gatsby's GraphQL data API
- Live example:

Filtering with GraphQL

Queried results can be filtered down with operators like `eq` (equals), `ne` (not equals), `in`, and `regex` on specified fields.

For this recipe, you'll need a Gatsby site with a collection of nodes to filter in the GraphQL data layer. All sites have some nodes like `allSitePage` created automatically: more can be added by installing source and transformer plugins like `gatsby-source-filesystem` and `gatsby-transformer-remark` in `gatsby-config.js` to produce `allMarkdownRemark`.

Prerequisites

- A Gatsby site
- Queryable fields prefixed with `all`, e.g. `allSitePage` or `allMarkdownRemark`

Directions

1. Run `gatsby develop` to start the development server.
2. Open the GraphiQL explorer in a browser tab at: `http://localhost:8000/___graphql`
3. Add a query in the editor using a field prefixed by 'all', like `allMarkdownRemark` (meaning that it will return a list of nodes)

```
{
  allMarkdownRemark {
    edges {
      node {
        frontmatter {
          title
          categories
        }
      }
    }
  }
}
```

4. Add a `filter` argument to the `allMarkdownRemark` field and give it an object with the fields you'd like to filter by. In this example, Markdown content is filtered by the `categories` attribute in frontmatter metadata. The next value is the operator: in this case `eq`, or equals, with a value of 'magical creatures'.

```
{
  allMarkdownRemark(filter: {frontmatter: {categories: {eq: "magical creatures"}}}) { // hig
    edges {
      node {
        frontmatter {
          title
          categories
        }
      }
    }
  }
}
```

5. Click the play button in the GraphiQL page. The data that matches the filter parameters should be returned, in this case only sourced Markdown files tagged with a category of 'magical creatures'.

Additional resources

- Gatsby GraphQL reference for filtering
- Complete list of possible operators
- Learn about nodes in Gatsby's GraphQL data API
- Live example:

GraphQL Query Aliases

You can rename any field in a GraphQL query with an alias.

If you would like to run two queries on the same datasource, you can use an alias to avoid a naming collision with two queries of the same name.

Directions

1. Run `gatsby develop` to start the development server.
2. Open the GraphiQL explorer in a browser tab at: `http://localhost:8000/___graphql`
3. Add a query in the editor using two fields of the same name like `allFile`

```
{
  allFile {
    totalCount
  }
  allFile {
    pageInfo {
      currentPage
    }
  }
}
```

4. Add the name you would like to use for any field before the name of the field in your GraphQL schema, separated by a colon. (E.g. `[alias-name]: [original-name]`)

```
{
  fileCount: allFile { // highlight-line
    totalCount
  }
  filePageInfo: allFile { // highlight-line
    pageInfo {
      currentPage
    }
  }
}
```

5. Click the play button in the GraphiQL page and 2 objects with alias names you provided should be output.

Additional resources

- Gatsby GraphQL reference for aliasing
- Live example:

GraphQL Query Fragments

GraphQL fragments are shareable chunks of a query that can be reused.

You might want to use them to share multiple fields between queries or to colocate a component with the data it uses.

Directions

1. Declare a `graphql` template string with a Fragment in it. The fragment should be made up of the keyword `fragment`, a name, the GraphQL type it is associated with (in this case of type `Site`, as demonstrated by `on Site`), and the fields that make up the fragment:

```
export const query = graphql`
  // highlight-start
  fragment SiteInformation on Site {
    title
    description
  }
  // highlight-end
`
```

2. Now, include the fragment in a query for a field of the type specified by the fragment. This includes those fields without having to declare them all independently:

```
export const pageQuery = graphql`
  query SiteQuery {
    site {
-     title
-     description
+   ...SiteInformation
    }
  }
`
```

Note: Fragments don't need to be imported in Gatsby. Exporting a query with a Fragment makes that Fragment available in *all* queries in your project.

Fragments can be nested inside other fragments, and multiple fragments can be used in the same query.

Additional resources

- Example repo using fragments
- Gatsby GraphQL reference for fragments
- Example repo with co-located data

Querying data client-side with `fetch`

Data doesn't only have to be queried at build time and remain solely static. You can query data at runtime the same way you can fetch data in a regular React app.

Prerequisites

- A Gatsby Site
- A page component, such as `index.js`

Directions

1. In a file with a React component defined, like a page in `src/pages` or a layout component, import React hooks for `useState` and `useEffect`.

```
import React, { useState, useEffect } from "react"
```

2. Inside the component, wrap a function to fetch data in a `useEffect` hook so it will asynchronously retrieve data when the component mounts in the browser client. Then, `await` the result with the `fetch` API, and call the set function from the `useState` hook (in this case `setStarsCount`) to save the state variable (`starsCount`) to the data returned from `fetch`.

```
import React, { useState, useEffect } from "react"

const IndexPage = () => {
  // highlight-start
  const [starsCount, setStarsCount] = useState(0)
  useEffect(() => {
    // get data from GitHub api
    fetch(`https://api.github.com/repos/gatsbyjs/gatsby`)
      .then(response => response.json()) // parse JSON from request
      .then(resultData => {
        setStarsCount(resultData.stargazers_count)
      }) // set data for the number of stars
  }, [])
  // highlight-end

  return (
    <section>
      // highlight-start
      <p>Runtime Data: Star count for the Gatsby repo {starsCount}</p>
      // highlight-end
    </section>
  )
}

export default IndexPage
```

Additional resources

- Guide on client-data fetching
- Live example site using this example