GraphQL Concepts

There are many options for loading data into React components. One of the most popular and powerful of these is a technology called GraphQL.

GraphQL was invented at Facebook to help product engineers *pull* needed data into React components.

GraphQL is a **q**uery **l**anguage (the *QL* part of its name). If you're familiar with SQL, it works in a very similar way. Using a special syntax, you describe the data you want in your component and then that data is given to you.

Gatsby uses GraphQL to enable page and StaticQuery components to declare what data they and their sub-components need. Then, Gatsby makes that data available in the browser when needed by your components.

Data from any number of sources is made queryable in one unified layer, a key part of the Gatsby building process:

Why is GraphQL so cool?

For a more in-depth look, read why Gatsby uses GraphQL.

- Eliminate frontend data boilerplate — no need to worry about requesting & waiting for data. Just ask for the data you need with a GraphQL query and it'll show up when you need it
- Push frontend complexity into queries — many data transformations can be done at *build-time* within your GraphQL queries
- It's the perfect data querying language for the often complex/nested data dependencies of modern applications
- Improve performance by removing data bloat — GraphQL enables you to select only the data you need, not whatever an API returns

What does a GraphQL query look like?

GraphQL lets you ask for the exact data you need. Queries look like JSON:

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

```
    }
  }
}
```

Which returns this:

```
{
  "site": {
    "siteMetadata": {
      "title": "A Gatsby site!"
    }
  }
}
```

A basic page component with a GraphQL query might look like this:

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

export default function Page({ data }) {
  return (
    <div>
      <h1>About {data.site.siteMetadata.title}</h1>
      <p>We're a very cool website you should return to often.</p>
    </div>
  )
}

export const query = graphql`
  query {
    site {
      siteMetadata {
        title
      }
    }
  }
`
```

The result of the query is automatically inserted into your React component on the `data` prop. GraphQL and Gatsby let you ask for data and then immediately start using it.

Note: To run GraphQL queries in non-page components you'll need to use Gatsby's Static Query feature.

Understanding the parts of a query

The following diagram shows a GraphQL query, with each word highlighted in a color corresponding to its name on the legend:

GraphQL query diagram

Query operation type The diagram marks the word `query` as the "Operation Type", for Gatsby's uses the only operation type you will deal with is `query`, this can be omitted from your queries if you prefer (like in the above example).

Operation name `SiteInformation` is marked as the "Operation Name", which is a unique name that you assign to a query yourself. This is similar to how you would name a function or a variable, and like a function this can be omitted if you would rather the query be anonymous.

Query fields The four words `site`, `id`, `siteMetadata`, and `title` are marked as "Fields". Any top-level fields – like `site` in the diagram – are sometimes referred to as **root level fields**, though the name doesn't signify functional significance as all fields in GraphQL queries behave the same.

How to learn GraphQL

Your experience developing with Gatsby might be the first time you've seen GraphQL! We hope you love it as much as we do and find it useful for all your projects.

When starting out with GraphQL, we recommend the following two tutorials:

- https://www.howtographql.com/
- https://graphql.org/learn/

The official Gatsby tutorial also includes an introduction to using GraphQL specifically with Gatsby.

How do GraphQL and Gatsby work together?

One of the great things about GraphQL is how flexible it is. People use GraphQL with many different programming languages and for web and native apps.

Most people run GraphQL on a server to respond live to requests for data from clients. You define a schema (a schema is a formal way of describing the shape of your data) for your GraphQL server and then your GraphQL resolvers retrieve data from databases and/or other APIs.

Gatsby uses GraphQL at *build-time* and *not* for live sites. This is unique, and it means you don't need to run additional services (e.g. a database and Node.js service) to use GraphQL for production websites.

Gatsby is a great framework for building apps so it's possible and encouraged to pair Gatsby's native build-time GraphQL with GraphQL queries running against a live GraphQL server from the browser.

Where does Gatsby's GraphQL schema come from?

Most usages of GraphQL involve manually creating a GraphQL schema.

Gatsby uses plugins which can fetch data from different sources. That data is used to automatically *infer* a GraphQL schema.

If you give Gatsby data that looks like this:

```
{
  "title": "A long long time ago"
}
```

Gatsby will create a schema that looks something like this:

```
title: String
```

This makes it possible to pull data from anywhere and immediately start writing GraphQL queries against your data.

This *can* cause confusion as some data sources allow you to define a schema even when there's not any data added for parts or all of the schema. If parts of the data haven't been added, then those parts of the schema might not be recreated in Gatsby.

Powerful data transformations

GraphQL enables another unique feature of Gatsby — it lets you control data transformations with arguments to your queries. Some examples follow.

Formatting dates

People often store dates like "2018-01-05" but want to display the date in some other form like "January 5th, 2018". One way of doing this is to load a date-formatting JavaScript library into the browser. Or, with Gatsby's GraphQL layer, you can do the formatting at query-time like:

```
{
  date(formatString: "MMMM Do, YYYY")
}
```

See the full list of formatting options by viewing our GraphQL reference page.

Markdown

Gatsby has *transformer* plugins which can transform data from one form to another. A common example is markdown. If you install `gatsby-transformer-remark`, then in your queries, you can specify if you want the transformed HTML version instead of markdown:

```
markdownRemark {
  html
}
```

Images

Gatsby has rich support for processing images. Responsive images are a big part of the modern web and typically involve creating 5+ sized thumbnails per photo. With Gatsby's `gatsby-transformer-sharp`, you can *query* your images for responsive versions. The query automatically creates all the needed responsive thumbnails and returns `src` and `srcSet` fields to add to your image element.

Combined with a special Gatsby image component, gatsby-plugin-image, you have a very powerful set of primitives for building sites with images.

This is what a component using `gatsby-plugin-image` looks like:

```jsx
import React from "react"
import { GatsbyImage } from "gatsby-plugin-image"
import { graphql } from "gatsby"

export default function Page({ data }) {
  return (
    <div>
      <h1>Hello gatsby-plugin-image</h1>
      <GatsbyImage image={data.file.childImageSharp.gatsbyImageData} />
    </div>
  )
}

export const query = graphql`
  query {
    file(relativePath: { eq: "blog/avatars/kyle-mathews.jpeg" }) {
      childImageSharp {
        # Specify the image processing specifications right in the query.
        # Makes it trivial to update as your page's design changes.
        gatsbyImageData(width: 125, height: 125, layout: FIXED)
      }
    }
  }
`
```

See also the following blog posts:

- Making Website Building Fun
- Image Optimization Made Easy with Gatsby.js

Advanced

Fragments

Notice that in the above example for querying images, we used `...GatsbyImageSharpFixed`, which is a GraphQL Fragment, a reusable set of fields for query composition. You can read more about them here.

If you wish to define your own fragments for use in your application, you can use named exports to export them in any JavaScript file, and they will be automatically processed by Gatsby for use in your GraphQL queries.

For example, if I put a fragment in a helper component, I can use that fragment in any other query:

```
export const markdownFrontmatterFragment = graphql`
  fragment MarkdownFrontmatter on MarkdownRemark {
    frontmatter {
      path
      title
      date(formatString: "MMMM DD, YYYY")
    }
  }
`
```

They can then be used in any GraphQL query after that!

```
query ($path: String!) {
  markdownRemark(frontmatter: { path: { eq: $path } }) {
    ...MarkdownFrontmatter
  }
}
```

It's good practice for your helper components to define and export a fragment for the data they need. For example, on your index page you might map over all of your posts to show them in a list.

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

export default function Home({ data }) {
  return (
    <div>
      <h1>Index page</h1>
      <h4>{data.allMarkdownRemark.totalCount} Posts</h4>
      {data.allMarkdownRemark.edges.map(({ node }) => (
        <div key={node.id}>
          <h3>
            {node.frontmatter.title} <span>- {node.frontmatter.date}</span>
          </h3>
```

```
      </div>
    ))}
    </div>
  )
}

export const query = graphql`
  query {
    allMarkdownRemark {
      totalCount
      edges {
        node {
          id
          frontmatter {
            title
            date(formatString: "DD MMMM, YYYY")
          }
        }
      }
    }
  }
`
```

If the index component becomes too large, you might want to refactor it into smaller components.

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

export default function IndexPost({ frontmatter: { title, date } }) {
  return (
    <div>
      <h3>
        {title} <span>- {date}</span>
      </h3>
    </div>
  )
}

export const query = graphql`
  fragment IndexPostFragment on MarkdownRemark {
    frontmatter {
      title
      date(formatString: "MMMM DD, YYYY")
    }
  }
`
```

Now, you can use the component together with the exported fragment in your index page.

```
import React from "react"
import IndexPost from "../components/IndexPost"
import { graphql } from "gatsby"

export default function Home({ data }) {
  return (
    <div>
      <h1>Index page</h1>
      <h4>{data.allMarkdownRemark.totalCount} Posts</h4>
      {data.allMarkdownRemark.edges.map(({ node }) => (
        <div key={node.id}>
          <IndexPost frontmatter={node.frontmatter} />
        </div>
      ))}
    </div>
  )
}

export const query = graphql`
  query {
    allMarkdownRemark {
      totalCount
      edges {
        node {
          id
          ...IndexPostFragment
        }
      }
    }
  }
`
```

Further reading

- Why Gatsby Uses GraphQL
- The Anatomy of a GraphQL Query

Getting started with GraphQL

- https://graphql.org/learn/
- https://www.howtographql.com/
- https://reactjs.org/blog/2015/05/01/graphql-introduction.html
- https://services.github.com/on-demand/graphql/

Advanced readings on GraphQL

- GraphQL specification
- Interfaces and Unions
- Relay Compiler (which Gatsby uses to process queries)