gatsby-source-graphql

Plugin for connecting arbitrary GraphQL APIs to Gatsby's GraphQL. Remote schemas are stitched together by declaring an arbitrary type name that wraps the remote schema Query type (`typeName` below), and putting the remote schema under a field of the Gatsby GraphQL query (`fieldName` below).

- Example website
- Example website source

This source plugin does **not** support incremental builds, cloud builds, and preview (on Gatsby Cloud). Please be aware that build times will be signficantly slower than regular source plugins as the size of your site goes past a hundred or so pages.

Install

`npm install gatsby-source-graphql`

How to use

If the remote GraphQL API needs authentication, you should pass environment variables to the build process, so credentials aren't committed to source control. We recommend using `dotenv` , which will then expose environment variables. Read more about dotenv and using environment variables here. Then we can *use* these environment variables via `process.env` and configure our plugin.

```
// In your gatsby-config.js
module.exports = {
  plugins: [
    // Simple config, passing URL
    {
      resolve: "gatsby-source-graphql",
      options: {
        // Arbitrary name for the remote schema Query type
        typeName: "SWAPI",
        // Field under which the remote schema will be accessible. You'll use this
in your Gatsby query
        fieldName: "swapi",
        // Url to query from
        url: "https://swapi-graphql.netlify.app/.netlify/functions/index",
      },
    },

    // Advanced config, passing parameters to apollo-link
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "GitHub",
        fieldName: "github",
        url: "https://api.github.com/graphql",
        // HTTP headers
        headers: {
          // Learn about environment variables: https://gatsby.dev/env-vars
          Authorization: `Bearer ${process.env.GITHUB_TOKEN}`,
        },
        // HTTP headers alternatively accepts a function (allows async)
        headers: async () => {
          return {
            Authorization: await getAuthorizationToken(),
          }
        },
        // Additional options to pass to node-fetch
        fetchOptions: {},
      },
    },

    // Advanced config, using a custom fetch function
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "GitHub",
        fieldName: "github",
        url: "https://api.github.com/graphql",
        // A `fetch`-compatible API to use when making requests.
        fetch: (uri, options = {}) =>
          fetch(uri, { ...options, headers: sign(options.headers) }),
      },
    },

    // Complex situations: creating arbitrary Apollo Link
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "GitHub",
        fieldName: "github",
        // Create Apollo Link manually. Can return a Promise.
        createLink: pluginOptions => {
          return createHttpLink({
            uri: "https://api.github.com/graphql",
            headers: {
              Authorization: `Bearer ${process.env.GITHUB_TOKEN}`,
            },
            fetch,
          })
        },
      },
    },
  ],
}
```

How to Query

```
{
  # This is the fieldName you've defined in the config
  swapi {
    allSpecies {
      name
    }
  }
  github {
    viewer {
      email
    }
  }
}
```

Schema definitions

By default, the schema is introspected from the remote schema. The schema is cached in the `.cache` directory, and refreshing the schema requires deleting the cache (e.g. by restarting `gatsby develop`).

To control schema consumption, you can alternatively construct the schema definition by passing a `createSchema` callback. This way you could, for example, read schema SDL or introspection JSON. When the `createSchema` callback is used, the schema isn't cached. `createSchema` can return a GraphQLSchema instance, or a Promise resolving to one.

```
const fs = require("fs")
const { buildSchema, buildClientSchema } = require("graphql")

module.exports = {
  plugins: [
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "SWAPI",
        fieldName: "swapi",
        url: "https://api.graphcms.com/simple/v1/swapi",

        createSchema: async () => {
          const json = JSON.parse(
            fs.readFileSync(`${__dirname}/introspection.json`)
          )
          return buildClientSchema(json.data)
        },
      },
    },
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "SWAPI",
        fieldName: "swapi",
        url: "https://api.graphcms.com/simple/v1/swapi",
        createSchema: async () => {
          const sdl = fs.readFileSync(`${__dirname}/schema.sdl`).toString()
          return buildSchema(sdl)
        },
      },
    },
  ],
}
```

Composing Apollo Links for production network setup

Network requests can fail, return errors or take too long. Use Apollo Link to add retries, error handling, logging and more to your GraphQL requests.

Use the plugin's `createLink` option to add a custom Apollo Link to your GraphQL requests.

You can compose different types of links, depending on the functionality you're trying to achieve. The most common links are:

- `@apollo/client/link/retry` for retrying queries that fail or time out
- `@apollo/client/link/error` for error handling
- `@apollo/client/link/http` for sending queries in http requests (used by default)

For more explanation of how Apollo Links work together, check out this Medium article: Productionizing Apollo Links.

Here's an example of using the HTTP link with retries (using @apollo/client/link/retry):

```
// gatsby-config.js
const { createHttpLink, from } = require(`@apollo/client`)
const { RetryLink } = require(`@apollo/client/link/retry`)

const retryLink = new RetryLink({
  delay: {
    initial: 100,
    max: 2000,
    jitter: true,
  },
  attempts: {
    max: 5,
    retryIf: (error, operation) =>
      Boolean(error) && ![500, 400].includes(error.statusCode),
  },
})

module.exports = {
  plugins: [
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "SWAPI",
        fieldName: "swapi",
        url: "https://api.graphcms.com/simple/v1/swapi",

        // `pluginOptions`: all plugin options
        //   (i.e. in this example object with keys `typeName`, `fieldName`, `url`,
`createLink`)
        createLink: pluginOptions =>
          from([retryLink, createHttpLink({ uri: pluginOptions.url })]),
      },
    },
  ],
}
```

Custom transform schema function (advanced)

It's possible to modify the remote schema, via a `transformSchema` option which customizes the way the default schema is transformed before it is merged on the Gatsby schema by the stitching process.

The `transformSchema` function gets an object argument with the following fields:

- schema (introspected remote schema)
- link (default link)
- resolver (default resolver)
- defaultTransforms (an array with the default transforms)
- options (plugin options)

The return value is expected to be the final schema used for stitching.

Below an example configuration that uses the default implementation (equivalent to not using the `transformSchema` option at all):

```
const { wrapSchema } = require(`@graphql-tools/wrap`)
const { linkToExecutor } = require(`@graphql-tools/links`)

module.exports = {
  plugins: [
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "SWAPI",
        fieldName: "swapi",
        url: "https://api.graphcms.com/simple/v1/swapi",
        transformSchema: ({
          schema,
          link,
          resolver,
          defaultTransforms,
          options,
        }) => {
          return wrapSchema(
            {
              schema,
              executor: linkToExecutor(link),
            },
            defaultTransforms
          )
        }
      },
    ]
  }
```

For details, refer to https://www.graphql-tools.com/docs/schema-wrapping.

An use case for this feature can be seen in this issue.

Refetching data

By default, `gatsby-source-graphql` will only refetch the data once the server is restarted. It's also possible to configure the plugin to periodically refetch the data. The option is called `refetchInterval` and specifies the timeout in seconds.

```
module.exports = {
  plugins: [
    // Simple config, passing URL
    {
      resolve: "gatsby-source-graphql",
      options: {
        // Arbitrary name for the remote schema Query type
        typeName: "SWAPI",
        // Field under which the remote schema will be accessible. You'll use this
in your Gatsby query
        fieldName: "swapi",
        // Url to query from
        url: "https://api.graphcms.com/simple/v1/swapi",

        // refetch interval in seconds
        refetchInterval: 60,
      },
    },
  ],
}
```

Performance tuning

By default, `gatsby-source-graphql` executes each query in a separate network request. But the plugin also supports query batching to improve query performance.

Caveat: Batching is only possible for queries starting at approximately the same time. In other words it is bounded by the number of parallel GraphQL queries executed by Gatsby (by default it is **4**).

Fortunately, we can increase the number of queries executed in parallel by setting the environment variable `GATSBY_EXPERIMENTAL_QUERY_CONCURRENCY` to a higher value and setting the `batch` option of the plugin to `true` .

Example:

```
cross-env GATSBY_EXPERIMENTAL_QUERY_CONCURRENCY=20 gatsby develop
```

With plugin config:

```
const fs = require("fs")
const { buildSchema, buildClientSchema } = require("graphql")

module.exports = {
  plugins: [
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "SWAPI",
        fieldName: "swapi",
        url: "https://api.graphcms.com/simple/v1/swapi",
        batch: true,
      },
    },
  ],
}
```

By default, the plugin batches up to 5 queries. You can override this by passing `dataLoaderOptions` and set a `maxBatchSize`:

```
const fs = require("fs")
const { buildSchema, buildClientSchema } = require("graphql")

module.exports = {
  plugins: [
    {
      resolve: "gatsby-source-graphql",
      options: {
        typeName: "SWAPI",
        fieldName: "swapi",
        url: "https://api.graphcms.com/simple/v1/swapi",
        batch: true,
        // See https://github.com/graphql/dataloader#new-dataloaderbatchloadfn--
options
        // for a full list of DataLoader options
        dataLoaderOptions: {
          maxBatchSize: 10,
        },
      },
    },
  ],
}
```

Having 20 parallel queries with 5 queries per batch means we are still running 4 batches in parallel.

Each project is unique so try tuning those two variables and see what works best for you. We've seen up to 5-10x speed-up for some setups.

How batching works

Under the hood `gatsby-source-graphql` uses DataLoader for query batching. It merges all queries from a batch to a single query that gets sent to the server in a single network request.

Consider the following example where both of these queries are run:

```
{
  query: `query(id: Int!) {
    node(id: $id) {
      foo
    }
  }`,
  variables: { id: 1 },
}
```

```
{
  query: `query(id: Int!) {
    node(id: $id) {
      bar
    }
  }`,
  variables: { id: 2 },
}
```

They will be merged into a single query:

```
{
  query: `
    query(
      $gatsby0_id: Int!
      $gatsby1_id: Int!
    ) {
      gatsby0_node: node(id: $gatsby0_id) {
        foo
      }
      gatsby1_node: node(id: $gatsby1_id) {
        bar
      }
    }
  `,
  variables: {
    gatsby0_id: 1,
    gatsby1_id: 2,
  }
}
```

Then `gatsby-source-graphql` splits the result of this single query into multiple results and delivers it back to Gatsby as if it executed multiple queries:

```
{
  data: {
    gatsby0_node: { foo: `foo` },
    gatsby1_node: { bar: `bar` },
  },
}
```

is transformed back to:

```
[
  { data { node: { foo: `foo` } } },
  { data { node: { bar: `bar` } } },
]
```

Note that if any query result contains errors the whole batch will fail.

Apollo-style batching

If your server supports apollo-style query batching you can also try HttpLinkDataLoader. Pass it to the `gatsby-source-graphql` plugin via the `createLink` option.

This strategy is usually slower than query merging but provides better error reporting.