Creating First Class Gatsby Source Plugins

Create Gatsby plugins that leverage Gatsby's most impactful native features like remote image optimization, caching, customized GraphQL schemas and node relationships, and more.

This monorepo serves as an example of a site using a first class source plugin to pull in data from a Node.js API. It is meant to show the 3 pieces that work together when building a source plugin: the API, the site, and the source plugin.

Setup

This monorepo uses yarn workspaces to manage the 3 individual projects:

- api: a Node.js API with in-memory data, and a Post and Author type, as well as support for subscriptions when Posts are mutated
- example-site: a barebones Gatsby site that implements the source plugin
- source-plugin: a plugin that uses several Gatsby APIs to source data from the API, create responsive/optimized images from remote locations, and link the nodes in the example site

To install dependencies for all projects run the install command in the root of the yarn workspace (which requires yarn to be installed):

```
yarn install
```

Note: if you aren't using yarn, you can navigate into each of the 3 folders and run ***`npm install`*** *instead*

Then you can run the api or example projects in separate terminal windows with the commands below.

For the API which runs at `http://localhost:4000`, use this command:

```
yarn workspace api start
```

And to run the example site with `gatsby develop` at `http://localhost:8000`, use this command:

```
yarn workspace example-site develop
```

Running the example site also runs the plugin because it is included in the site's config. You'll see output in the console for different functionality and then can open up the browser to `http://localhost:8000` to see the site.

Developing and Experimenting

You can open up `http://localhost:4000` with the API running, which will load a GraphQL Playground, which is a GraphQL IDE (like GraphiQL, that Gatsby runs at `http://localhost:8000/___graphql`) for running queries and mutations on the data from the API.

You can test a query like this to see data returned:

```
query {
  posts {
    id
    slug
  }
}
```

This query will return the IDs for all posts in the API. You can copy one of these IDs and provide it as an argument to a mutation to update information about that post.

You can run 3 different mutations from the GraphQL Playground (at `http://localhost:4000`): `createPost`, `updatePost`, and `deletePost`. These methods would mimic CRUD operations happening on the API of the data source like a headless CMS. An example `updatePost` mutation is outlined below.

When you run a mutation on a post, a subscription event is published, which lets the plugin know it should respond and update nodes. The following mutation can be copied into the left side of the GraphQL playground so long as you replace "post-id" with a value returned for an ID from a query (like the one above).

```
mutation {
  updatePost(id: "post-id", description: "Some data!") {
    id
    slug
    description
  }
}
```

The website's homepage will update with any changes while the source plugin is subscribed to changes, which is when the `preview: true` is provided in the example site's `gatsby-config`.

You can also optionally listen for subscription events with this query in the playground which will display data when a mutation is run:

```
subscription {
  posts {
    id
    description
  }
}
```

A similar subscription is registered when the plugin is run, so you can also see subscription events logged when the plugin is running.