import { Announcement } from "gatsby-interface"

In this tutorial, you'll create your own source plugin that will gather data from an API. The plugin will source data, optimize remote images, and create foreign key relationships between data sourced by your plugin.

## What is a source plugin?

Source plugins "source" data from remote or local locations into what Gatsby calls nodes. This tutorial uses a demo API so that you can see how the data works on both the frontend and backend, but the same principles apply if you would like to source data from another API.

At a high-level, a source plugin:

- Ensures local data is synced with its source and is 100% accurate.
- Creates nodes with accurate media types, human-readable types, and accurate contentDigests.
- Links nodes & creates relationships between them.
- Lets Gatsby know when nodes are finished sourcing so it can move on to processing them.

A source plugin is a regular npm package. It has a `package.json` file, with optional dependencies, as well as a `gatsby-node.js` file where you implement Gatsby's Node APIs. Read more about files Gatsby looks for in a plugin or creating a generic plugin.

## Why create a source plugin?

Source plugins convert data from any source into a format that Gatsby can process. Your Gatsby site can use several source plugins to combine data in interesting ways.

There may not be an existing plugin for your data source, so you can create your own.

**Please Note:** If your data is local i.e. on your file system and part of your site's repo, then you generally don't want to create a new source plugin. Instead you want to use gatsby-source-filesystem which handles reading and watching files for you. You can then use transformer plugins like gatsby-transformer-yaml to make queryable data from files.

## How to create a source plugin

### Overview

The plugin in this tutorial will source blog posts and authors from the demo API, link the posts and authors, and take image URLs from the posts and optimize them automatically. You'll be able to configure your plugin in your site's `gatsby-config.js` file and write GraphQL queries to access your plugin's data.

This tutorial builds off of an existing Gatsby site and some data. If you want to follow along with this tutorial, you can find the codebase inside the examples folder of the Gatsby repository. Once you clone this code, make sure to delete the `source-plugin` and `example-site` folders. Otherwise, the tutorial steps will already be completed.

#### An example API request

To see the API in action, you can run it locally by navigating into the `api` folder, installing dependencies with `npm install`, and starting the server with `npm start`. You will then be able to navigate to a GraphQL playground running at `http://localhost:4000`. This is a GraphQL server running in Node.js and is *separate from Gatsby*, this server could be replaced with a different backend or data source and the patterns in this tutorial would remain the same. Other possible examples could be a REST API, local files, or even a database, so long as you can access data it can be sourced.

If you paste the following query into the left side of the window and press the play button, you should see data for posts with their IDs and descriptions returned:

```
query {
  posts {
    id
    description
  }
}
```

This data is an example of the data you will source with your plugin.

#### Plugin behavior

Your plugin will have the following behavior:

- Make an API request to the demo API.
- Convert the data in the API response to Gatsby's node system.
- Link the nodes together so you can query for an author on each post.
- Accept plugin options to customize how your plugin works.
- Optimize images from Unsplash URLs so they can be used with `gatsby-image`.

### Set up projects for plugin development

You'll need to set up an example site and create a plugin inside it to begin building.

#### Set up an example site

Create a new Gatsby site with the `gatsby new` command, based on the hello world starter.

```
gatsby new example-site https://github.com/gatsbyjs/gatsby-starter-hello-world
```

This site generated by the `new` command is where the plugin will be installed, giving you a place to test the code for your plugin.

#### Set up a source plugin

Create a new Gatsby plugin with the `gatsby new` command, this time based on the plugin starter.

```
gatsby new source-plugin https://github.com/gatsbyjs/gatsby-starter-plugin
```

This will create your plugin in a separate project from your example site, but you could also include it in your site's `plugins` folder.

Your plugin starts with a few files from the starter, which can be seen in the snippet below:

```
/example-site
/source-plugin
├── .gitignore
├── gatsby-browser.js
├── gatsby-node.js
├── gatsby-ssr.js
├── index.js
```

```
├── package.json
└── README.md
```

The biggest changes will be in **`gatsby-node.js`**. This file is where Gatsby expects to find any usage of the [Gatsby Node APIs](). These allow customization/extension of default Gatsby settings affecting pieces of the site build process. All the logic for sourcing data will live in this file.

#### Install your plugin in the example site

You need to install your plugin in the site to be able to test that your code is running. Gatsby only knows to run plugins that are included in its `gatsby-config.js` file. Open up the `gatsby-config.js` file in the `example-site` and [add your plugin using `require.resolve`]() . If you decide to publish your plugin it can be installed with an `npm install <plugin-name>` and including the name of the plugin in the config instead of `require.resolve` .

```
module.exports = {
  plugins: [require.resolve(`../source-plugin`)],
}
```

*You can include the plugin by using its name if you are using [npm link or yarn workspaces]() or place your `source-plugin` in [`example-site/plugins`]() instead of being in a folder a step above and using `require.resolve` .*

You can now navigate into the `example-site` folder and run `gatsby develop` . You should see a line in the output in the terminal that shows your plugin loaded:

```
$ gatsby develop
success open and validate gatsby-configs - 0.033s
success load plugins - 0.074s
Loaded gatsby-starter-plugin // highlight-line
success onPreInit - 0.016s
...
```

If you open the `gatsby-node.js` file in your `source-plugin` folder, you will see the `console.log` that produces that output in the terminal.

```
exports.onPreInit = () => console.log("Loaded gatsby-starter-plugin")
```

### Source data and create nodes

Data is sourced in the `gatsby-node.js` file of source plugins or Gatsby sites. Specifically, it's done by calling a Gatsby function called `createNode` inside of the `sourceNodes` API in the `gatsby-node.js` file.

#### Create nodes inside of `sourceNodes` with the `createNode` function

Open up the `gatsby-node.js` file in the `source-plugin` project and add the following code to create nodes from a hardcoded array of data :

```
// constants for your GraphQL Post and Author types
const POST_NODE_TYPE = `Post`
```

```
exports.sourceNodes = async ({
  actions,
  createContentDigest,
  createNodeId,
  getNodesByType,
}) => {
  const { createNode } = actions

  const data = {
    posts: [
      { id: 1, description: `Hello world!` },
      { id: 2, description: `Second post!` },
    ],
  }

  // loop through data and create Gatsby nodes
  data.posts.forEach(post =>
    createNode({
      ...post,
      id: createNodeId(`${POST_NODE_TYPE}-${post.id}`),
      parent: null,
      children: [],
      internal: {
        type: POST_NODE_TYPE,
        content: JSON.stringify(post),
        contentDigest: createContentDigest(post),
      },
    })
  )

  return
}
```

This code creates Gatsby nodes that are queryable in a site. The following bullets break down what is happening in the code:

- You implemented Gatsby's `sourceNodes` API, which Gatsby will run as part of its bootstrap process, and pulled out some Gatsby helpers (like `createContentDigest` and `createNodeId`) to facilitate creating nodes.
- You provided the required fields for the node like creating a node ID and a content digest (which Gatsby uses to track dirty nodes—or nodes that have changed). The content digest should include the whole content of the item (`post`, in this case).
- Then you stored some data in an array and looped through it, calling `createNode` on each post in the array.

If you run the `example-site` with `gatsby develop`, you can now open up `http://localhost:8000/___graphql` and query your posts with this query:

```
query {
  allPost {
    nodes {
```

```
      id
      description
    }
  }
}
```

The problem with this data is that it is *not coming from the API*, it is hardcoded into an array. The declaration of the `data` array needs to be updated to pull data from a different location.

### Caching data between runs

Some operations like fetching data from an endpoint can be performance heavy or time-intensive. In order to improve the experience of developing with your source plugin, you can leverage the Gatsby cache to store data between runs of `gatsby develop` or `gatsby build`.

You access the `cache` in Gatsby Node APIs and use the `set` and `get` functions to store and retrieve data as JSON objects.

```
exports.onPostBuild = async ({ cache }) => {
  await cache.set(`key`, `value`)
  const cachedValue = await cache.get(`key`)
  console.log(cachedValue) // logs `value`
}
```

The above snippet shows a contrived example for the `cache`, but it can be used in more sophisticated cases to reduce the time it takes to run your plugin. For example, by caching a timestamp, you can use it to fetch solely the data that has been updated since the last time data was fetched from the source:

```
exports.sourceNodes = async ({ cache }) => {
  // get the last timestamp from the cache
  const lastFetched = await cache.get(`timestamp`)

  // pull data from some remote source using cached data as an option in the request
  const data = await fetch(
    `https://remotedatasource.com/posts?lastUpdated=${lastFetched}`
  )
  // ...
}

exports.onPostBuild = async ({ cache }) => {
  // set a timestamp at the end of the build
  await cache.set(`timestamp`, Date.now())
}
```

This can reduce the time it takes repeated data fetching operations to run if you are pulling in large amounts of data for your plugin. Existing plugins like `gatsby-source-contentful` generate a token that is sent with each request to only return new data.

You can read more about the cache API, other types of plugins that leverage the cache, and example open source plugins that use the cache in the build caching guide.

## Querying and sourcing data from a remote location

You can query data from any location to source at build time using functions and libraries like Node.js's built-in `http.get`, `axios`, or `node-fetch`. This tutorial uses a GraphQL client so that the source plugin can support GraphQL subscriptions when it fetches data from the demo API, and can proactively update your data in the site when information on the API changes.

### Adding dependencies

You'll use several modules from npm to making fetching data with GraphQL easier. Install them in the `source-plugin` project with:

```
npm install @apollo/client graphql subscriptions-transport-ws ws
```

*Note: The libraries used here are specifically chosen so that the source plugin can support GraphQL subscriptions. You can fetch data the same way you would in any other Node.js app or however you are most comfortable.*

Open your `package.json` file after installation and you'll see the packages have been added to a `dependencies` section at the end of the file.

### Configure an Apollo client to fetch data

Import the handful of Apollo packages that you installed to help set up an Apollo client in your plugin:

```
// highlight-start
const { ApolloClient, InMemoryCache, gql, split, HttpLink } =
require("@apollo/client")
const { WebSocketLink } = require("@apollo/client/link/ws")
const { getMainDefinition } = require("@apollo/client/utilities")
const WebSocket = require("ws")
const fetch = require("node-fetch")
// highlight-end

const POST_NODE_TYPE = `Post`

exports.sourceNodes = async ({
// ...
```

Then you can copy this code that sets up the necessary pieces of the Apollo client and paste it after your imports:

```
// ... imports
const WebSocket = require("ws")

const POST_NODE_TYPE = `Post`

// highlight-start
const client = new ApolloClient({
  link: split(
    ({ query }) => {
      const definition = getMainDefinition(query)
      return (
```

```
        definition.kind === "OperationDefinition" &&
        definition.operation === "subscription"
      )
    },
    new WebSocketLink({
      uri: `ws://localhost:4000`, // or `ws://gatsby-source-plugin-api.glitch.me/`
      options: {
        reconnect: true,
      },
      webSocketImpl: WebSocket,
    }),
    new HttpLink({
      uri: "http://localhost:4000", // or `https://gatsby-source-plugin-
api.glitch.me/`
      fetch,
    })
  ),
  cache: new InMemoryCache(),
})
// highlight-end

exports.sourceNodes = async ({
// ...
```

You can read about each of the packages that are working together in Apollo's docs. The end result is creating a `client` that you can use to call methods like `query` to get data from the source it's configured to work with. In this case, that is `http://localhost:4000` where you should have the API running.

**Query data from the API**

Now you can replace the hardcoded data in the `sourceNodes` function with a GraphQL query:

```
  exports.sourceNodes = async ({
    actions,
    createContentDigest,
    createNodeId,
    getNodesByType,
  }) => {
    const { createNode, touchNode, deleteNode } = actions

-   const data = {
-     posts: [
-       { id: 1, description: `Hello world!` },
-       { id: 2, description: `Second post!` },
-     ],
-   }
+   const { data } = await client.query({
+     query: gql`
+       query {
+         posts {
+           id
+           description
```

```
+         }
+       }
+     `,
+   })

    // ...
```

Now you're creating nodes based on data coming from the API. Neat! However, only the `id` and `description` fields are coming back from the API and being saved to each node, so add the rest of the fields to the query so that the same data is available to Gatsby.

This is also a good time to add data to your query so that it also returns authors.

```
const { data } = await client.query({
  query: gql`
    query {
      posts {
        id
        description
        // highlight-start
        slug
        imgUrl
        imgAlt
        author {
          id
          name
        }
      }
      authors {
        id
        name
      }
      // highlight-end
    }
  `,
})
```

With the new data, you can also loop through the authors to create Gatsby nodes from them by adding another loop to `sourceNodes`:

```
const POST_NODE_TYPE = `Post`
const AUTHOR_NODE_TYPE = `Author` // highlight-line

exports.sourceNodes = async ({
  actions,
  createContentDigest,
  createNodeId,
  getNodesByType,
}) => {
  const { createNode, touchNode, deleteNode } = actions
```

```
  const { data } = await client.query({
    query: gql`
      query {
        posts {
          id
          slug
          description
          imgUrl
          imgAlt
          author {
            id
            name
          }
        }
        authors {
          id
          name
        }
      }
    `,
  })

  // loop through data returned from the api and create Gatsby nodes for them
  data.posts.forEach(post =>
    createNode({
      ...post,
      id: createNodeId(`${POST_NODE_TYPE}-${post.id}`), // hashes the inputs into an
ID
      parent: null,
      children: [],
      internal: {
        type: POST_NODE_TYPE,
        content: JSON.stringify(post),
        contentDigest: createContentDigest(post),
      },
    })
  )
  // highlight-start
  data.authors.forEach(author =>
    createNode({
      ...author,
      id: createNodeId(`${AUTHOR_NODE_TYPE}-${author.id}`), // hashes the inputs
into an ID
      parent: null,
      children: [],
      internal: {
        type: AUTHOR_NODE_TYPE,
        content: JSON.stringify(author),
        contentDigest: createContentDigest(author),
      },
    })
  )
```

```
  // highlight-end

  return
}
```

At this point you should be able to run `gatsby develop` in your `example-site`, open up GraphiQL at `http://localhost:8000/___graphql` and query both posts and authors.

```
query {
  allPost {
    nodes {
      id
      description
      imgUrl
    }
  }
  allAuthor {
    nodes {
      id
      name
    }
  }
}
```

## Working with data received from remote sources

### Setting media and MIME types

Each node created by the filesystem source plugin includes the raw content of the file and its *media type*.

A **media type** (also **MIME type** and **content type**) is an official way to identify the format of files/content that are transmitted via the internet, e.g. over HTTP or through email. You might be familiar with other media types such as `application/javascript`, `audio/mpeg`, `text/html`, etc.

Each source plugin is responsible for setting the media type for the nodes it creates. This way, source and transformer plugins can work together easily.

This is not a required field -- if it's not provided, Gatsby will infer the type from data that is sent -- but it's how source plugins indicate to transformers that there is "raw" data the transformer can further process.

It also allows plugins to remain small and focused. Source plugins don't have to have opinions on how to transform their data: they can set the `mediaType` and push that responsibility to transformer plugins instead.

For example, it's common for services to allow you to add content in Markdown format. If you pull that Markdown into Gatsby and create a new node, what then? How would a user of your source plugin convert that Markdown into HTML they can use in their site? You would create a node for the Markdown content and set its `mediaType` as `text/markdown` and the various Gatsby Markdown transformer plugins would see your node and transform it into HTML.

This loose coupling between the data source and the transformer plugins allow Gatsby site builders to assemble complex data transformation pipelines with little work on their (and your (the source plugin author)) part.

### Optimize remote images

Each node of post data has an `imgUrl` field with the URL of an image on Unsplash. You could use that URL to load images on your site, but they will be large and take a long time to load. You can optimize the images with your source plugin so that a site using your plugin already has data for `gatsby-plugin-image` ready to go!

You can read about how to use the Gatsby Image plugin if you are unfamiliar with it.

#### Create remote file node from a URL

To create optimized images from URLs, `File` nodes for image files need to be added to your site's data. Then, you can install `gatsby-plugin-sharp` and `gatsby-transformer-sharp` which will automatically find image files and add the data needed for `gatsby-plugin-image`.

Start by installing `gatsby-source-filesystem` in the `source-plugin` project:

```
npm install gatsby-source-filesystem
```

Now in your plugin's `gatsby-node.js` file, you can implement a new API, called `onCreateNode`, that gets called every time a node is created. You can check if the node created was one of your `Post` nodes, and if it was, create a file from the URL on the `imgUrl` field.

Import the `createRemoteFileNode` helper from `gatsby-source-filesystem`, which will download a file from a remote location and create a `File` node for you.

```
const {
  ApolloClient,
  InMemoryCache,
  split,
  HttpLink,
} = require("@apollo/client")
const { WebSocketLink } = require("@apollo/client/link/ws")
const { getMainDefinition } = require("@apollo/client/utilities")
const fetch = require("node-fetch")
const gql = require("graphql-tag")
const WebSocket = require("ws")
// highlight-start
const { createRemoteFileNode } = require(`gatsby-source-filesystem`)
// highlight-end
```

Then export a new function `onCreateNode`, and call `createRemoteFileNode` in it whenever a node of type `Post` is created:

```
// called each time a node is created
exports.onCreateNode = async ({
  node, // the node that was just created
  actions: { createNode, createNodeField },
  createNodeId,
  getCache,
}) => {
  if (node.internal.type === POST_NODE_TYPE) {
```

```
    const fileNode = await createRemoteFileNode({
      // the url of the remote image to generate a node for
      url: node.imgUrl,
      parentNodeId: node.id,
      createNode,
      createNodeId,
      getCache,
    })

    if (fileNode) {
      createNodeField({ node, name: 'localFile', value: fileNode.id })
    }
  }
}
```

This code is called every time a node is created, e.g. when `createNode` is invoked. Each time it is called in the `sourceNodes` step, the condition will check if the node was a `Post` node. Since those are the only nodes with an image associated with them, that is the only time images need to be optimized. Then a remote node is created, if it's successful, the `fileNode` is returned. The next few lines are important:

```
if (fileNode) {
  createNodeField({ node, name: 'localFile', value: fileNode.id })
}
```

By using `createNodeField` you're extending the existing node and place a new field named `localFile` under the `fields` key.

<Announcement style={{marginBottom: "1.5rem"}}>

**Note:** Do not mutate the `node` directly and use `createNodeField` instead. Otherwise the change won't be persisted and you might see inconsistent data. This behavior changed with Gatsby 4, read the migration guide to learn more.

In the previous step you only defined the `fileNode.id` as a `value` but at this time Gatsby can't just yet resolve this to the `fileNode` (and susequently the image) itself. Therefore, you'll need to create a foreign-key relationship between the `Post` node and the respective `File` node. Use the `createSchemaCustomization` API to define this relationship:

```
exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions

  createTypes(`
    type Post implements Node {
      localFile: File @link(from: "fields.localFile")
    }
  `)
}
```

**Note**: You can use schema customization APIs to create these kinds of connections between nodes as well as sturdier and more strictly typed ones.

You now can query the image like this:

```
query {
  allPost {
    nodes {
      id
      localFile {
        id
        relativePath
      }
    }
  }
}
```

At this point you have created local image files from the remote locations and associated them with your posts, but you still need to transform the files into optimized versions.

#### Transform `File` nodes with sharp plugins

sharp plugins make optimization of images possible at build time.

Install `gatsby-plugin-sharp`, `gatsby-plugin-image`, and `gatsby-transformer-sharp` in the `example-site` (*not* the plugin):

```
npm install gatsby-plugin-sharp gatsby-transformer-sharp gatsby-plugin-image
```

Then include the plugins in your `gatsby-config`:

```
module.exports = {
  plugins: [
    require.resolve(`../source-plugin`),
    // highlight-start
    `gatsby-plugin-image`,
    `gatsby-plugin-sharp`,
    `gatsby-transformer-sharp`,
    // highlight-end
  ],
}
```

By installing the sharp plugins in the site, they'll run after the source plugin and transform the file nodes and add fields for the optimized versions at `childImageSharp`. The transformer plugin looks for `File` nodes with extensions like `.jpg` and `.png` to create optimized images and creates the GraphQL fields for you.

Now when you run your site, you will also be able to query a `childImageSharp` field on the `post.localFile`:

```
query {
  allPost {
    nodes {
      localFile {
        // highlight-start
```

```
        childImageSharp {
          gatsbyImageData
        }
        // highlight-end
      }
    }
  }
}
```

With data available, you can now query optimized images to use with the `gatsby-plugin-image` component in a site!

### Create foreign key relationships between data

To link the posts to the authors, Gatsby needs to be aware that the two are associated, and how. You have already implemented one example of this when Gatsby inferred a connection between a `localFile` and the remote file from Unsplash.

The best approach for connecting related data is through customizing the GraphQL schema. By implementing the `createSchemaCustomization` API, you can specify the exact shape of a node's data. While defining that shape, you can optionally link a node to other nodes to create a relationship.

Copy this code and add it to the `source-plugin` in the `gatsby-node.js` file:

```
exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions
  createTypes(`
    type Post implements Node {
      id: ID!
      slug: String!
      description: String!
      imgUrl: String!
      imgAlt: String!
      # create relationships between Post and Author nodes
      author: Author @link(from: "author.name" by: "name")
      # Created in previous step
      localFile: @link(from: "fields.localFile")
    }
    type Author implements Node {
      id: ID!
      name: String!
    }
  `)
}
```

The `author: Author @link(from: "author.name" by: "name")` line tells Gatsby to look for the value on the `Post` node at `post.author.name` and relate it with an `Author` node with a matching `name`. This demonstrates the ability to link using more than just an ID.

Now running the site will allow you to query authors from the post nodes!

```
query {
  allPost {
    nodes {
      id
      // highlight-start
      author {
        name
      }
      // highlight-end
    }
  }
}
```

### Using data from the source plugin in a site

In the `example-site`, you can now query data from pages.

Add a file at `example-site/src/pages/index.js` and copy the following code into it:

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

export default ({ data }) => (
  <>
    <h1>Posts</h1>
    <section
      style={{
        display: `grid`,
        gridTemplateColumns: `repeat( auto-fit, minmax(250px, 1fr) )`,
        gridGap: 16,
      }}
    >
      {data.allPost.nodes.map(post => (
        <div
          style={{
            display: `flex`,
            flexDirection: `column`,
            padding: 16,
            border: `1px solid #ccc`,
          }}
        >
          <h2>{post.slug}</h2>
          <span>By: {post.author.name}</span>
          <p>{post.description}</p>
          <GatsbyImage
            image={post.localFile?.childImageSharp?.gatsbyImageData}
            alt={post.imgAlt}
          />
        </div>
      ))}
```

```
    </section>
  </>
)

export const query = graphql`
  {
    allPost {
      nodes {
        id
        slug
        description
        imgAlt
        author {
          id
          name
        }
        localFile {
          childImageSharp {
            gatsbyImageData(formats: [AUTO, WEBP, AVIF])
          }
        }
      }
    }
  }
`
```

This code uses a page query to fetch all posts and provide them to the component in the `data` prop at build time. The JSX code loops through the posts so they can be rendered to the DOM.

### Using plugin options to customize plugin usage

You can pass options into a plugin through a `gatsby-config.js` file. Update the code where your plugin is installed in the `example-site`, changing it from a string, to an object with a `resolve` and `options` key.

```
module.exports = {
  plugins: [
    // highlight-start
    {
      resolve: require.resolve(`../source-plugin`),
      options: {
        previewMode: true,
      },
    },
    // highlight-end
    `gatsby-plugin-sharp`,
    `gatsby-transformer-sharp`,
  ],
}
```

Now the options you designated (like `previewMode: true`) will be passed into each of the Gatsby Node APIs like `sourceNodes`, making options accessible inside of Gatsby APIs. Add an argument called `pluginOptions` to

your `sourceNodes` function.

```
exports.sourceNodes = async (
  { actions, createContentDigest, createNodeId, getNodesByType },
  pluginOptions // highlight-line
) => {
  const { createNode, touchNode, deleteNode } = actions
  // highlight-start
  console.log(pluginOptions.previewMode) // true
  // highlight-end

  // ... other code removed for brevity
}
```

Options can be a good way of providing conditional paths to logic that you as a plugin author want to provide or limit. Read the Configuring Plugin usage with Plugin Options guide to learn how to add validation to your plugin options.

### Improve plugin developer experience by enabling faster sync

One challenge when developing locally is that a developer might make modifications in a remote data source, like a CMS, and then want to see how it looks in the local environment. Typically they will have to restart the `gatsby develop` server to see changes. In order to improve the development experience of using a plugin, you can reduce the time it takes to sync between Gatsby and the data source by enabling faster synchronization of data changes. The best way to do this is by adding **event-based syncing**.

Some data sources keep event logs and are able to return a list of objects modified since a given time. If you're building a source plugin, you can store the last time you fetched data using the cache and then only sync down nodes that have been modified since that time. `gatsby-source-contentful` is an example of a source plugin that does this.

### Enabling Content Sync

If you would like to add Content Sync to your source plugin here but aren't sure what it is learn more about Content Sync here. To enable this feature in your source plugin you will need to make sure that your data source (or CMS) also works with Content Sync.

#### Content Sync Source Plugin changes

The source plugin needs to create node manifests using the `unstable_createNodeManifest` action.

#### Identifying which nodes need manifests

The first thing you'll want to do is identify which nodes you'll want to create a node manifest for. These will typically be nodes that you can preview, entry nodes, top level nodes, etc. An example of this could be a blog post or an article, any node that can be the "owner" of a page. A good place to call this action is whenever you call `createNode`.

An easy way to keep track of your manifest logic is to parse it out into a different util function. Either inside the `createNodeManifest` util or before you call it you'll need to vet which nodes you'll want to create manifests for.

```
import { createNodeManifest } from "./utils.js"
exports.sourceNodes = async (
```

```
  { actions }
) => {
  // sourcing data...
  const { unstable_createNodeManifest, createNode } = actions

  nodes.forEach(node => {
    // highlight-start
    const gatsbyNode = createNode(node)

    const nodeIsEntryNode = `some condition`
    if (nodeIsEntryNode) {
      createNodeManifest({
        entryItem: node,
        entryNode: gatsbyNode,
        project,
        unstable_createNodeManifest,
      })
    }
    // highlight-end
  })
}
```

**Check for support**

At the moment you'll only want to create node manifests for preview content and because this is a newer API, we'll need to check if the Gatsby version supports `unstable_createNodeManifest` .

```
export function createNodeManifest({
  entryItem, // the raw data source/cms content data
  project,   // the cms project data
  entryNode, // the Gatsby node
  unstable_createNodeManifest,
}) {
  // highlight-start
  // This env variable is provided automatically on Gatsby Cloud hosting
  const isPreview = process.env.GATSBY_IS_PREVIEW === `true`

  const createNodeManifestIsSupported =
    typeof unstable_createNodeManifest === `function`

  const shouldCreateNodeManifest = isPreview && createNodeManifestIsSupported
  // highlight-end

  if (shouldCreateNodeManifest) {
    // create manifest...
  }
}
```

**Call `unstable_createNodeManifest`**

Next we will build up the `manifestId` and call `unstable_createNodeManifest` . The `manifestId` needs to be created with information that comes from the CMS **NOT** Gatsby (the CMS will need to create the exact same

manifest), which is why we use the `entryItem` id as opposed to the `entryNode` id. This `manifestId` must be uniquely tied to a specific revision of specific content. We use the CMS project space (you may not need this), the id of the content, and finally the timestamp that it was updated at.

```
export function createNodeManifest({
  // ...
}) {
  // ...

  if (shouldCreateNodeManifest) {
    // highlight-start
    const updatedAt = entryItem.updatedAt
    const manifestId = `${project.id}-${entryItem.id}-${updatedAt}`

    unstable_createNodeManifest({
      manifestId,
      node: entryNode,
      updatedAtUTC: updatedAt,
    })
    // highlight-end
  }
}
```

**Warn if no support**

Lastly we'll want to give our users a good experience and give a warning if they're using a version of Gatsby that does not support Content Sync

```
// highlight-start
let warnOnceForNoSupport = false
// highlight-end

export function createNodeManifest({
  // ...
}) {
  // ...

  if (shouldCreateNodeManifest) {
    // ...
  } else if (
    // highlight-start
    // it's helpful to let users know if they're using an outdated Gatsby version so they'll upgrade for the best experience
    isPreview && !createNodeManifestIsSupported && !warnOnceForNoSupport
  ) {
    console.warn(
      `${sourcePluginName}: Your version of Gatsby core doesn't support Content Sync (via the unstable_createNodeManifest action). Please upgrade to the latest version to use Content Sync in your site.`
    )
    // This is getting called for every entry node so we don't want the console logs to get cluttered
```

```
    warnOnceForNoSupport = true
    // highlight-end
  }
}
```

**Content Sync data source (or CMS) changes**

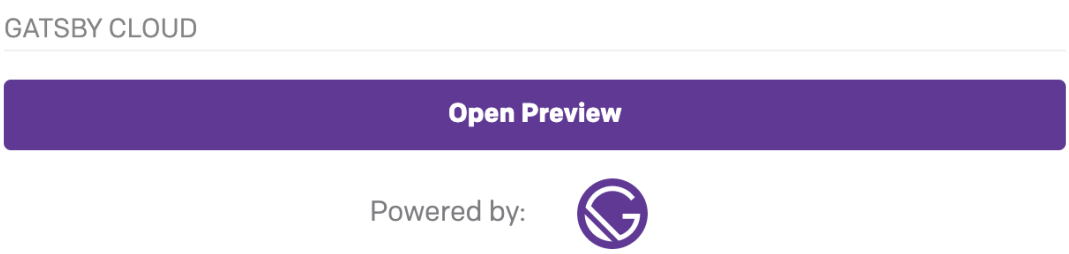
The CMS will need to send a preview webhook to Gatsby Cloud when content is changed and open the Content Sync waiting room. Follow along to learn how to implement it on the CMS side.

**Using the Gatsby preview extension**

You will need to create a button in your CMS that does the following:

1. `POST` to the preview webhook url in Gatsby Cloud
2. Open the Content Sync waiting room

The button might look something like this:




**Configuration**

You will need to store the Content Sync URL from a given Gatsby Cloud site as a CMS extension option. This will look something like `https://gatsbyjs.com/content-sync/<siteId>`. This is often done in the CMS plugin extension configuration, this will differ from CMS to CMS depending on how extension settings are stored.

You will also need to store the preview webhook URL. This might also be stored in the plugin extension settings, but often is stored in a separate CMS webhooks settings page if your CMS supports webhooks already. Find out how to get that webhook url here

Both of these need to be user configurable in the CMS.

NOTE: The Content Sync URL can be found in the same place as the webhook url in the Gatsby Cloud site settings.

**Building up the manifest id**

Recall that we need to create a matching manifest id in the CMS AND the Gatsby plugin. Whenever content is saved we can build up a new manifest id that will look the same as the manifest id we created in the source plugin

```
const manifestId = `${project.id}-${entryItem.id}-${updatedAt}`
```

In the CMS extension, we should have access to

- the project id (if the CMS uses one)
- the content id
- the timestamp that the content was updated at (or some other piece of data that is tied to a very specific state of saved content)

#### Enabling "Eager Redirects" with Content ID's

Eager Redirects is a Content Sync feature which causes the user to be redirected to their site frontend as soon as possible. When they first preview a piece of content, they will stay in the Content Sync loading screen until their preview is ready. On subsequent previews of that same piece of content, they will be redirected as soon as the page loads. This is done by storing a "content ID" in local storage. The content ID should be a unique identifier for that piece of content which is consistent across all previews.

```
const contentId = project.id
```

This content ID should be appended to the end of the Content Sync URL. See the sections below for more information.

#### Starting the preview build (optional)

If the CMS does not handle this part automatically we will need to tell Gatsby cloud to build a preview by `POST`ing to the Gatsby Cloud preview build webhook url.

#### Opening the Content Sync waiting room

Once we've built a `manifestId` and `POST`ed to the preview build webhook url, we need to open a new tab/window with a modified version of the Content Sync URL. You get that by grabbing the Content Sync URL you stored in the CMS extension earlier and appending the Gatsby source plugin name, the content ID, and the content's `manifestId` that you just created, `https://gatsbyjs.com/content-sync/<siteId>/<sourcePluginName>/<manifestId>/<contentId>/` .

#### Useful Content Sync development tips

Here are some things to keep in mind and some "gotchas" depending on how the CMS acts.

- Inside the CMS, sometimes you will need to wait to make sure you have the correct `updatedAt` timestamp as some CMS may take a second to update their backend and then wait for the change to propagate to the frontend. While others will immediately update the frontend and then propagate that to the backend. You will need the *most* up to date timestamp when opening the Content Sync UI waiting room.
- Make sure that a preview webhook is being sent to Gatsby Cloud after the content is edited, whether it's before you press the "Open Preview" button or the "Open Preview" is the trigger that sends the webhook.
- While developing, you can set the Gatsby `VERBOSE` env variable to `"true"` to see additional logs that will help you debug what's happening in the source plugin.
- When you click the "Open Preview" button in the CMS the `manifestId` in the URL should match the `manifestId` that the source plugin creates from that revision.
- The node manifests get written out in the `public` dir of your gatsby site, so you can check to manifests on your local disk `/public/__node-manifests/<sourcePluginName>/<manifestId>.json` or you can navigate directly to that piece of content `https://<your-domain>/__node-manifests/<sourcePluginName>/<manifestId>`

### Enabling Image CDN support

Image CDN is a feature on Gatsby Cloud that provides edge network image processing by waiting to perform image processing until the very first user visit to a page. The processed image is then cached for super quick fetching on all subsequent user views. Enabling it will also speed up local development builds and production builds on other deployment platforms because images from your CMS or data source will only be downloaded if they are used in a created Gatsby page.

You can learn more about it in the announcement blogpost Image CDN: Lightning Fast Image Processing for Gatsby Cloud

#### Integration with source plugins

Starting with `gatsby@4.10`, Image CDN and its helper functions are available inside `gatsby` and `gatsby-plugin-utils`. In addition to the `RemoteFile` interface you can also use the `addRemoteFilePolyfillInterface` and `polyfillImageServiceDevRoutes` functions to enable Image CDN support down to Gatsby 2 inside your plugin.

#### `createSchemaCustomization` node API additions

To add support to a source plugin, you will need to create a new GraphQL object type that implements the `Node` and `RemoteFile` interfaces.

```
schema.buildObjectType({
  name: `YourImageAssetNodeType`,
  fields: {
    // .. your fields
  },
  interfaces: [`Node`, `RemoteFile`],
})
```

It is also recommended that you add a polyfill to provide support back through Gatsby 2. To do so, wrap the `buildObjectType` call with the `addRemoteFilePolyfillInterface` polyfill like so:

```
import { addRemoteFilePolyfillInterface } from "gatsby-plugin-utils/polyfill-remote-file"

addRemoteFilePolyfillInterface(
  schema.buildObjectType({
    name: `YourImageAssetNodeType`,
    fields: {
      // your fields
    },
    interfaces: [`Node`, `RemoteFile`],
  }),
  {
    schema,
    actions,
    // schema and actions are arguments on the `createSchemaCustomization` API
  }
)
```

Implementing the `RemoteFile` interface adds the correct fields to your new GraphQL type and adds the necessary resolvers to handle the type. `RemoteFile` holds the following properties:

- `mimeType: String!`
- `filename: String!`
- `filesize: Int`
- `width: Int`
- `height: Int`
- `publicUrl: String!`

- `resize(width: Int, height: Int, fit: enum): String`
- `gatsbyImage(args): String`

You might notice that `width`, `height`, `resize`, and `gatsbyImage` can be null. This is because the `RemoteFile` interface can also handle assets other than images, like PDF's.

The string returned from `gatsbyImage` is intended to work seamlessly with Gatsby Image Component just like `gatsbyImageData` does.

#### `sourceNodes` node API additions

When creating nodes, you must add some fields to the node itself to match what the `RemoteFile` interface expects. You will need `url`, `mimeType`, `filename` as mandatory fields. When you have an image type, `width` and `height` are required as well. The optional fields are `placeholderUrl` and `filesize`. `placeholderUrl` will be the url used to generate blurred or dominant color placeholder so it should contain `%width%` and `%height%` url params if possible.

```
const assetNode = {
  id: `an id`,
  internal: {
    type: `PrefixAsset`,
  },
  url: `${file.url}`,
  placeholderUrl: `${file.url}?w=%width%&h=%height%`,
  mimeType: file.contentType,
  filename: file.fileName,
  width: file.details?.image?.width,
  height: file.details?.image?.height,
}
```

#### `onCreateDevServer` node API

Add the polyfill, `polyfillImageServiceDevRoutes`, to ensure that the development server started with `gatsby develop` has the routes it needs to work with Image CDN.

```
import { polyfillImageServiceDevRoutes } from "gatsby-plugin-utils/polyfill-remote-file"

export const onCreateDevServer = ({ app }) => {
  polyfillImageServiceDevRoutes(app)
}
```

Now you're all set up to use Image CDN! 🤗

## Publishing a plugin

Don't publish this particular plugin to npm or the Gatsby Plugin Library, because it's just a sample plugin for the tutorial. However, if you've built a local plugin for your project, and want to share it with others, npm allows you to publish your plugins. Check out the npm docs on How to Publish & Update a Package for more info.

**Please Note:** Once you have published your plugin on npm, don't forget to edit your plugin's `package.json` file to include info about your plugin. If you'd like to publish a plugin to the Gatsby Plugin Library (please do!), please follow these steps.

## Summary

You've written a Gatsby plugin that:

- can be configured with an entry in your `gatsby-config.js` file
- requests data from an API
- pulls the API data into Gatsby's node system
- allows the data to be queried with GraphQL
- optimizes images from a remote location automatically
- links data types with a customized GraphQL schema
- updates new data without needing to restart your Gatsby site

Congratulations!

## Additional resources

- Example repository with all of this code implemented