This doc goes over the high level areas of this plugin, roughly what they do, how they're related, and where to find the corresponding code.

Historical Info

This plugin was a ground-up rewrite of `gatsby-source-wordpress@^3.0.0` which used the WordPress REST API to source data. `gatsby-source-wordpress@^4.0.0` uses WPGraphQL to source data and is an entirely different plugin with no shared code. Work for `^4.0.0` was initially done in the `gatsby-source-wordpress-experimental` repo before being moved to the Gatsby monorepo.

Inspired by this plugin, the Gatsby GraphQL Toolkit was written to make creating source plugins for GraphQL API's easier. Ideally this plugin would use the toolkit but since it didn't exist when this plugin was created that wasn't possible. There are also features this plugin has which the toolkit doesn't (yet) have, so we would need to update the toolkit before making this plugin use it. This would be a large amount of effort for a small amount of returned value. In the best case scenario the plugin would work exactly the same as it does now but we would need to do likely months of work to get to that point.

I initially wrote this plugin in JS and later it was partially ported to TS. As a result there are a mix of JS and TS files. When I was initially planning this plugin, the Gatsby core team collectively made a decision that Gatsby core should be written in TS and any plugins should be written in JS to keep community contributions from requiring the knowledge of TS. This plugin turned out to be one of the largest Gatsby plugin codebases and I later decided it 100% needs TS. If you want to convert some JS files to TS please feel welcome to do so! We will accept TS conversion PR's.

The file you're reading was created many months after the first stable version of this plugin was released - so this is a non-exhaustive explanation of the architecture. This doc is most likely incomplete. If you notice something is missing, please take a crack at adding it here, or if you don't know much about the part you're documenting open a Github discussion so we can hash it out together.

— @TylerBarnes

`gatsby-node.ts` Steps

In `src/gatsby-node.ts` a helper (`runApisInSteps`) is being used to run different "steps" of the codebase one after another for each Gatsby Node API. Many parts of the codebase count on something else happening at an earlier point, so `runApisInSteps` is an easy way to visualize that.

`src/gatsby-node.ts` is the entry point for 99.999% of the plugin (`src/gatsby-browser.ts` only imports 1 css file) so it's a good jumping off point for looking at different areas of the plugin.

Each "step" is in it's own file in `src/steps` .

Query Generation / Remote Schema Ingestion

Before we can source data from WPGraphQL we need to generate GraphQL queries to source that data. We do that by making an introspection query to WPGraphQL and then using the response to generate queries using a custom query builder (`src/steps/ingest-remote-schema/build-queries-from-introspection`).

This logic runs during the Gatsby Node API `createSchemaCustomization` .

See `src/steps/ingest-remote-schema` .

Good to know

- As we're aware of what will be a future Gatsby node, the queries can be generated to only fetch data we wont already have. Connections from one node to other nodes only fetch the id(s).
- When a field with a selection set is queried more than 1 time, a fragment is automatically generated for it to keep query size smaller.
- Currently we fetch connection id's on both sides (for example `User.posts[].id` as well as `Post.author.id`) which can result in some amount of overfetching.
- In the future we should add an API to mark some field as being able to be resolved entirely via Gatsby without fetching any data. Since we have `Post.author.id` we don't need to fetch `User.posts[].id` since we can query for all posts that have the current user node id as the author.
- Unnecessary overfetching also occurs as the plugin doesn't know which fields are being used in the Gatsby site, so we fetch all WPGQL data that's available. In `gatsby develop` this is necessary as we don't know what data the developer will want, but in `gatsby build` we know exactly what queries are being made.
- We should automatically fetch only the fields that are queried for in cold builds and data updates outside of `gatsby develop` .
- We should have some basic algorithm to determine query complexity (or use WPGraphQL's once it's available) and automatically split up queries into multiple queries when they're too large. Internally we can already store multiple queries.

Schema Customization

Using the same introspection query we used in query generation, we use the remote schema to generate the WP/Gatsby schema to work with the Gatsby node model and within the parameters of any related plugin options.

For many different types of fields, we have a field transformer (`src/steps/create-schema-customization/transform-fields`) which transforms the remote schema introspection for those fields into type/field definitions that Gatsby's schema customization layer understands. All resolvers happen on the Gatsby side as an automatic replication of what the resolvers in WPGraphQL are doing. This is only for fields with no input arguments. We currently don't (and probably always wont) have a way to automatically carry over input arguments from WPGraphQL.

This logic runs during the Gatsby Node API `createSchemaCustomization` .

See `src/steps/create-schema-customization` .

Interactions between Schema Customization and Query Generation

Plugin options that impact both of these areas are under the `schema` section.

Fetched Types are Queryable Types

As queries are generated we store up a list of which types and fields have been queried. This list is used when customizing the Gatsby schema to omit any fields or types which weren't queried due to plugin options or internal logic.

Automatic Field Prefixing

In the case that a Union or Interface type has multiple different types that contain the same fields, fields are auto-prefixed with the typename + fieldname during query generation. Later during schema customization, there is resolver logic to account for this situation. Without this, many interface and union types would be unqueryable as the fields are conflicting and GQL can't differentiate between them. `[Typename].[fieldName]` must always resolve to a single type (considering a union or interface as a single type).

Schema Caching

Remote schema ingestion caches itself, diffing an md5 of the last remote schema it saw vs the current remote schema on each update (`src/steps/ingest-remote-schema/diff-schemas.js`). Anytime these are different it will re-generate all queries. When this happens, we will also re-run schema customization. In production this will cause the plugin to re-source all nodes, in development it will only update the schema and log out a warning to run `gatsby clean` if the schema update included a data change. Without this each production data update would be 10 - 30 seconds slower and development schema changes could cause the plugin to constantly re-source all data resulting in 5min+ wait times each time you modified your WPGraphQL schema.

Sourcing nodes

Using the queries we generated earlier, nodes are fetched via WPGraphQL using cursor pagination (`src/steps/source-nodes/fetch-nodes/fetch-nodes.js`). Some plugin options affect how this logic behaves, mainly the `Type.limit` , `schema.requestConcurrency` , and `schema.perPage` options. There isn't currently any request retry logic built into node sourcing. This is because cursor pagination is somewhat fragile in the context of Gatsby needing to source every node that exists. Cursor pagination is a long chain of requests and any request within that chain which fails prevents anything further down the chain from being requested. This severely limits the request concurrency within a single node type (to 1 concurrent request) and it also means we can't retry a more resource intensive query later while still processing more queries. In the future we should re-architect this so that WPGatsby will let us fetch lists of node id's 10k at a time (these requests will still be cursor paginated). Once we have all the id's for every node in WPGrapHQL we can construct queries that ask for a certain list of node's by ID. This will let us ratchet up the request concurrency and more cleanly retry failed requests without blocking

anything. Requests that continually fail can be automatically retried on the end of the queue after moving on to other requests or adapted to use less resources (by splitting the list of nodes for 1 request into 2 requests).

As nodes are fetched they're analyzed via regexp to find connection ID's to MediaItem nodes (which are WPGraphQL File nodes). See `src/steps/source-nodes/fetch-nodes/fetch-referenced-media-items.js`. After all other node types are sourced, we use these MediaItem ID's to fetch lists of them at a time. This allows us to only fetch MediaItems that are actually in use on the site (many WP admins will upload 5-10x more files than they need to). Since we have all the ID's upfront we don't need to use cursor pagination for these requests which allows us to run much more crafty retry logic. We can retry failed requests on the end of the queue, which some servers seem to like better than retrying the same thing after a small delay. In my (admittedly rudimentary) experiments with retrying MediaItem requests on the end of the request queue instead of immediately it significantly improved the reliability of sourcing very large WP sites with 20k+ nodes, and significantly sped up sourcing times for smaller sites since I could increase the request concurrency. We should investigate this more in the future and abstract this logic to be used with any request, not just MediaItem requests, since this could speed up node sourcing too.

See `src/steps/source-nodes`.

Compatibility api (DX and security-ish feature)

Since this plugin relies on the PHP WP plugins WPGatsby and WPGraphQL being installed and activated in the WordPress site, we can't immediately assume that the code for the source plugin will have the right versions of these plugins in the future as more features are added and as they've been added in the past. To solve this problem a remote compatibility API was added. See `src/supported-remote-plugin-versions.ts` for the version ranges of each WP plugin the current version of the source plugin supports.

See `areRemotePluginVersionsSatisfied` in `src/steps/check-plugin-requirements.ts` for where this logic runs. WPGatsby exposes an endpoint allowing us to send version ranges for WPGraphQL and WPGatsby to ask if the remote plugin versions are within this range. The reason we don't just expose the versions of WPGraphQL and WPGatsby directly is that hackers could scan the internet for sites with vulnerable versions of these plugins. Since we can only send a version range (which doesn't include specific patch versions), hackers can't be sure wether or not a vulnerable version has been patched, making it harder to target sites with vulnerable plugin versions installed.

We're using rematch which is a redux wrapper

Rematch makes using redux a bit cleaner. If I was to do it again I would use a state machine library instead, but this is how things are. Rematch has multiple stores which are called models and live in `src/models/index.ts`. We're currently using an old version of rematch as they changed their API fairly significantly and we don't have a compelling reason to upgrade.

Gatsby node api helpers/actions are stored in local redux (not gatsby redux)

At the beginning of each Node API `setGatsbyApiToState` (see above) is called to store the Gatsby Node API helpers in our Rematch model. This makes it easier to use Gatsby actions and helpers in deeply nested functions without passing them all the way down through each level.

Caching

All caching in the plugin is in 3 parts:

- Remote schema changes (schema MD5 diffing)
- ActionMonitor (WPGatsby change events)

- Hard caching files and data (for improved local dev DX)

Remote schema changes (schema MD5 diffing)

This happens in `src/steps/ingest-remote-schema/diff-schemas.js`. If the schema MD5 changes in production we refetch all data from WordPress as we can't be sure wether the new schema includes new or changed data. In development this isn't as critical and refetching everything can be really annoying so we only update the schema and log out a warning about running `gatsby clean` if data changed during the schema update.

ActionMonitor (WPGatsby change events)

After an initial cold build, the current timestamp is stored in cache. On any subsequent build this timestamp is sent via a gql query to WPGraphQL/WPGatsby/WordPress to ask what's changed since the last time data was fetched. See `src/steps/source-nodes/update-nodes/fetch-node-updates.js` and `src/steps/source-nodes/index.ts`. A list of change events is fetched and the source plugin loops through them and processes CREATE/UPDATE and DELETE events (see `src/steps/source-nodes/update-nodes/wp-actions`). WordPress itself doesn't store these events so there is a lot of custom code/logic in WPGatsby that hooks into various WordPress events and stores data in the WP db. See https://github.com/gatsbyjs/wp-gatsby/tree/master/src/ActionMonitor. After each update, a new timestamp is stored to later get change events since that time.

Hard caching files and data (for improved local dev DX)

As Gatsby eagerly clears the cache (being that it doesn't have as much context on individual sources as source plugins do) "hard" caching options have been added to this plugin. There is an option for hard caching files and another for hard caching data. These options cache data outside of the Gatsby cache and they are considered experimental API's. There's no garuntee of data validity with them. Their main purpose is to improve local development DX so that installing an NPM package or updating gatsby-config.js or gatsby-node.js doesn't cause the source plugin to refetch hundreds or thousands of images or nodes. Searching the project for `hardCacheMediaFiles` and `hardCacheData` will lead you to areas where this caching logic is implemented.

Basic Auth

Basic Auth options are added in `src/steps/source-nodes/create-nodes/create-remote-media-item-node.js` and `src/steps/source-nodes/create-nodes/create-remote-file-node/index.js`. This option is intended for use with server-level authentication, not WP level authentication. The reason for this is Gatsby data should be considered public data. Anything not public should not be exposed to Gatsby as it could be accidentally leaked via GraphQL queries or the `/__graphql` endpoint of a running Preview instance. This option is intended to allow you to lock down your `/graphql` WPGraphQL endpoint so it can only be requested with some credentials. See https://github.com/gatsbyjs/gatsby/blob/master/packages/gatsby-source-wordpress/docs/features/security.md for more info.

Debugging options

There are a lot of debugging options to print out different info about the process. Check the debugging section in plugin options docs https://github.com/gatsbyjs/gatsby/blob/master/packages/gatsby-source-wordpress/docs/plugin-options.md#debug. These are scattered throughout the codebase.

Plugin options schema and documentation generation

The plugin options schema is defined in `src/steps/declare-plugin-options-schema.ts`. On each `yarn build`, `yarn generate-plugin-options-docs` is also run. This npm script runs `generate-plugin-options-docs.js` which uses the plugin options schema to generate the plugin options docs (in `docs/plugin-options.md`).

`gatsby develop` DX features

When developing a site locally, this plugin periodically checks for data or schema events in WP and reacts to them automatically, updating data/schema on the fly while you're working. See `src/steps/source-nodes/update-nodes/content-update-interval.js`.

WPGatsby

The WPGatsby plugin stores and exposes WP change events for Gatsby to query and then update data/schema with. It also adds some additional fields to the WPGraphQL schema (like `schemaMd5` and compatibility api fields) which the source plugin requires to function. See https://github.com/gatsbyjs/wp-gatsby for more info.

Preview

The code for Preview lives partially in WPGatsby (see above) and partially in this plugin. There is a lot that goes into previews functioning since WPGatsby has it's own preview loader logic which talks back and forth with the Gatsby process from WP. See `src/steps/preview/preview.md` for a more detailed explanation. All preview logic in this plugin lives in `src/steps/preview` and `src/steps/source-nodes/index.ts`.

File processing

In this plugin we have 2 related node types for files, the `MediaItem` node type which contains meta information about the media item file, and `File` which is Gatsby's file node type (it lives on `MediaItem.localFile`). See `src/steps/source-nodes/create-nodes/create-remote-media-item-node.js`.

`MediaItem` nodes are only fetched if they're referenced via a GraphQL connection on another node. This is done so that we only fetch files we might need since admins often upload many more images than they use as they're editing content. See `src/steps/source-nodes/fetch-nodes/fetch-referenced-media-items.js`.

Local `File` nodes are fetched as a side effect of fetching `MediaItem` nodes via the `beforeChangeNode` plugin option. See the `WpMediaItem` default option in `src/models/gatsby-api.ts` (currently on line 218).

Since we have a list of all media item's that are in use by WPGraphQL ID's, we don't need to paginate through media items to fetch them. This allows us to retry failed requests on the end of the request queue (which increases the success rate during failures vs retrying in place) and it also allows us to parellelize data and file requests at any concurrency level we want. See `src/steps/source-nodes/fetch-nodes/fetch-referenced-media-items.js`.

In some cases it's not desireable to fetch all local `File` nodes on `MediaItem` nodes. A user might only want to fetch some or none of them so there is a `lazyNodes` option which causes `File` nodes to be fetched in GraphQL resolvers instead of during the source nodes api. See `src/models/gatsby-api.ts` (currently on line 226) and `src/steps/create-schema-customization/transform-fields/transform-object.js` (currently on line 73).

HTML processing

HTML is processed in node data via regexp find/replaces on the stringified JSON of each node. This is done for performance since very complex data structures will mean a lot of looping and recursively running checks and regexp replaces on each individual field at any arbitrary depth. See `src/steps/source-nodes/create-nodes/process-node.js`.

The following are processed:

- anchor links that point at WP are made into relative links
- file links (css background-image, anchor links to files, etc) are converted to static Gatsby files
- image tags are converted to Gatsby images with static files
- custom regexp find/replaces can replace any arbitrary strings

During the above processes the plugin will fetch MediaItem nodes (if it can find them by url) and/or File nodes (either by the MediaItem.sourceUrl field or by the url in html).

Non-node root fields

In addition to sourcing nodes this plugin sources any root fields (acf options for example) that it reasonably can. Any fields which require input args are automatically skipped. These fields are re-fetched on every data update since we can't store WPGatsby events about data that isn't attached to a node. See `src/steps/source-nodes/create-nodes/fetch-and-create-non-node-root-fields.js`.