> *This used to be called GraphQL Node Types Creation and should contain info about schema inference and inference > metadata*
>
> *This documentation isn't up to date with the latest schema customization changes.*
>
> *Outdated areas are:*
>
> - *the `inferObjectStructureFromNodes` function doesn't exist anymore*
> - *`setFieldsOnGraphQLNodeType` has been deprecated due to the new `createTypes` action*
> - *file node creation has been moved away from `setFileNodeRootType`*
>
> *You can help by making a PR to update this documentation.*

Gatsby creates a GraphQLObjectType for each distinct `node.internal.type` that is created during the source-nodes phase. Find out below how this is done.

GraphQL Types for each type of node

When running a GraphQL query, there are a variety of fields that you will want to query. Let's take an example, say we have the below query:

```
{
  file( relativePath: {  eq: `blogs/my-blog.md` } ) {
    childMarkdownRemark {
      frontmatter: {
        title
      }
    }
  }
}
```

When GraphQL runs, it will query all `file` nodes by their relativePath and return the first node that satisfies that query. Then, it will filter down the fields to return by the inner expression. I.e `{ childMarkdownRemark ... }`. The building of the query arguments is covered by the Inferring Input Filters doc. This section instead explains how the inner filter schema is generated (it must be generated before input filters are inferred).

During the sourceNodes phase, let's say that gatsby-source-filesystem ran and created a bunch of `File` nodes. Then, different transformers react via onCreateNode, resulting in children of different `node.internal.type` s being created.

There are 3 categories of node fields that we can query.

Fields on the created node object

```
node {
  relativePath,
  extension,
  size,
  accessTime
}
```

Child/Parent

```
node {
  childMarkdownRemark,
  childrenPostsJson,
  children,
  parent
}
```

fields created by setFieldsOnGraphQLNodeType

```
node {
  publicURL
}
```

Each of these categories of fields is created in a different way, explained below.

gqlType Creation

The Gatsby term for the GraphQLObjectType for a unique node type, is `gqlType`. GraphQLObjectTypes are objects that define the type name and fields. The field definitions are created by the createNodeFields function in `build-node-types.js`.

An important thing to note is that all gqlTypes are created before their fields are inferred. This allows fields to be of types that haven't yet been created due to their order of compilation. This is accomplished by the use of `fields` being a function (basically lazy functions).

The first step in inferring GraphQL Fields is to generate an `exampleValue`. It is the result of merging all fields of all nodes of the type in question. This `exampleValue` will therefore contain all potential field names and values, which allows us to infer each field's types. The logic to create it is in getExampleValues.

With the exampleValue in hand, we can use each of its key/values to infer the Type's fields (broken down by the 3 categories above).

Fields on the created node object

Fields on the node that were created directly by the source and transform plugins. E.g. for `File` type, these would be `relativePath`, `size`, `accessTime` etc.

The creation of these fields is handled by the `inferObjectStructureFromNodes` function in infer-graphql-type.js. Given an object, a field could be in one of 3 sub-categories:

1. It involves a mapping in gatsby-config.js
2. Its value is a foreign key reference to some other node (ends in `___NODE`)
3. It's a plain object or value (e.g. String, number, etc)

Mapping field

Mappings are explained in the gatsby-config.js docs. If the object field we're generating a GraphQL type for is configured in the gatsby-config mapping, then we handle it specially.

Imagine our top level Type we're currently generating fields for is `MarkdownRemark.frontmatter`. And the field we are creating a GraphQL field for is called `author`. And, that we have a mapping setup of:

```
mapping: {
  "MarkdownRemark.frontmatter.author": `AuthorYaml.name`,
},
```

The field generation in this case is handled by [`inferFromMapping`]. The first step is to find the type that is mapped to. In this case, `AuthorYaml`. This is known as the `linkedType`. That type will have a field to link by. In this case `name`. If one is not supplied, it defaults to `id`. This field is known as `linkedField`

Now we can create a GraphQL Field declaration whose type is `AuthorYaml` (which we look up in list of other `gqlTypes`). The field resolver will get the value for the node (in this case, the author string), and then search through the React nodes until it finds one whose type is `AuthorYaml` and whose `name` field matches the author string.

Foreign Key reference (`___NODE`)

If not a mapping field, it might instead end in `___NODE`, signifying that its value is an ID that is a foreign key reference to another node in Redux. Check out the [Source Plugin Tutorial] for how this works from a user point of view. Behind the scenes, the field inference is handled by [`inferFromFieldName`].

This is actually quite similar to the mapping case above. We remove the `___NODE` part of the field name. E.g. `author___NODE` would become `author`. Then, we find our `linkedNode`. I.e given the example value for `author` (which would be an ID), we find its actual node in Redux. Then, we find its type in processed types by its `internal.type`. Note, that also like in mapping fields, we can define the `linkedField` too. This can be specified via `nodeFieldname___NODE___linkedFieldName`. E.g. for `author___NODE___name`, the linkedField would be `name` instead of `id`.

Now we can return a new GraphQL Field object, whose type is the one found above. Its resolver searches through all Redux nodes until it finds one with the matching ID. As usual, it also creates a [page dependency], from the query context's path to the node ID.

If the foreign key value is an array of IDs, then instead of returning a Field declaration for a single field, we return a `GraphQLUnionType`, which is a union of all the distinct linked types in the array.

Plain object or value field

If the field was not handled as a mapping or foreign key reference, then it must be a normal every day field. E.g. a scalar, string, or plain object. These cases are handled by [`inferGraphQLType`].

The core of this step creates a GraphQL Field object, where the type is inferred directly via the result of `typeof`. E.g. `typeof(value) === 'boolean'` would result in type `GraphQLBoolean`. Since these are simple values, resolvers are not defined (graphql-js takes care of that for us).

If however, the value is an object or array, we recurse, using [`inferObjectStructureFromNodes`] to create the GraphQL fields.

In addition, Gatsby creates custom GraphQL types for `File` ([types/type-file.js]) and `Date` ([types/type-date.js]). If the value of our field is a string that looks like a filename or a date (handled by [should-infer] functions), then we return the appropriate custom type.

Child/Parent fields

Child fields creation

Let's continue with the `File` type example. There are many transformer plugins that implement `onCreateNode` for `File` nodes. These produce `File` children that are of their own type. E.g. `markdownRemark`, `postsJson`.

Gatsby stores these children in Redux as IDs in the parent's `children` field. And then stores those child nodes as full Redux nodes themselves (see Node Creation for more). E.g. for a File node with two children, it will be stored in the Redux `nodes` namespace as:

```
{
  `id1`: { type: `File`, children: [`id2`, `id3`], ...other_fields },
  `id2`: { type: `markdownRemark`, ...other_fields },
  `id3`: { type: `postsJson`, ...other_fields }
}
```

An important note here is that we do not store a distinct collection of each type of child. Rather we store a single collection that they're all packed into. The benefit of this is that we can create a `File.children` field that returns all children, regardless of type. The downside is that the creation of fields such as `File.childMarkdownRemark` and `File.childrenPostsJson` is more complicated. This is what createNodeFields does.

Another convenience Gatsby provides is the ability to query a node's `child` or `children`, depending on whether a parent node has 1 or more children of that type.

child resolvers

When defining our parent `File` gqlType, createNodeFields will iterate over the distinct types of its children, and create their fields. Let's say one of these child types is `markdownRemark`. Let's assume there is only one `markdownRemark` child per `File`. Therefore, its field name is `childMarkdownRemark`. Now, we must create its GraphQL Resolver.

```
resolve(node, args, context, info)
```

The resolve function will be called when we are running queries for our pages. A query might look like:

```
query {
  file( relativePath { eq: "blog/my-blog.md" } ) {
    childMarkdownRemark { html }
  }
}
```

To resolve `file.childMarkdownRemark`, we take the node we're resolving, and filter over all of its `children` until we find one of type `markdownRemark`, which is returned. Remember that `children` is a collection of IDs. So as part of this, we look up the node by ID in Redux too.

But before we return from the resolve function, remember that we might be running this query within the context of a page. If that's the case, then whenever the node changes, the page will need to be rerendered. To record that fact, we call createPageDependency with the node ID and the page, which is a field in the `context` object in the resolve function signature.

parent field

When a node is created as a child of some node (parent), that fact is stored in the child's `parent` field. The value of which is the ID of the parent. The GraphQL resolver for this field looks up the parent by that ID in Redux and returns it. It also creates a page dependency, to record that the page being queried depends on the parent node.

Plugin fields

These are fields created by plugins that implement the setFieldsOnGraphQLNodeType API. These plugins return full GraphQL Field declarations, complete with type and resolve functions.

File types

As described in plain object or value field, if a string field value looks like a file path, then we infer `File` as the field's type. The creation of this type occurs in type-file.js setFileNodeRootType(). It is called just after we have created the GqlType for `File` (only called once).

It creates a new GraphQL Field Config whose type is the just created `File` GqlType, and whose resolver converts a string into a File object. Here's how it works:

Say we have a `data/posts.json` file that has been sourced (of type `File`), and then the gatsby-transformer-json transformer creates a child node (of type `PostsJson`)

```
;[
  {
    id: "1685001452849004065",
    text: "Venice is 👌",
    image: "images/BdiU-TTFP4h.jpg",
  },
]
```

Notice that the image value looks like a file. Therefore, we'd like to query it as if it were a file, and get its relativePath, accessTime, etc.

```
{
  postsJson(id: { eq: "1685001452849004065" }) {
    image {
      relativePath
      accessTime
    }
  }
}
```

The File type resolver takes care of this. It gets the value (`images/BdiU-TTFP4h.jpg`). It then looks up this node's root NodeID via Node Tracking which returns the original `data/posts.json` file. It creates a new filename by concatenating the field value onto the parent node's directory.

I.e `data` + `images/BdiU-TTFP4h.jpg` = `data/images/BdiU-TTFP4h.jpg` .

And then finally it searches Redux for the first `File` node whose path matches this one. This is our proper resolved node. We're done!