Schema Generation

Once the nodes have been sourced and transformed, the next step is to generate the GraphQL Schema. Gatsby Schema is different from many regular GraphQL schemas in that it combines plugin or user defined schema information with data inferred from the nodes' shapes. The latter is called *schema inference*. Users or plugins can explicitly define the schema, in whole or in part, using the schema customization API. Usually, every node will get a GraphQL Type based on its `node.internal.type` field. When using Schema Customization, all types that implement the `Node` interface become GraphQL Node Types and thus get root level fields for accessing them.

GraphQL Compose

Schema creation is done using the `graphql-compose` library. GraphQL Compose is a toolkit for creating schemas programmatically. It has great tools to add types and fields in an iterative manner. Gatsby does lots of processing and schema generation, so a library like this fits the use case perfectly.

You can create a schema in GraphQL Compose by adding types to a Schema Composer - an intermediate object that holds all the schema types inside itself. After all modifications are done, the composer is converted into a regular GraphQL Schema.

1. Schema inference

Every time a node is created, Gatsby will generate *inference metadata* for it. Metadata for each node can be merged with other metadata, meaning that it's possible to derive the least generic possible schema for a particular node type. Inference metadata can also detect if some data is conflicting. In most cases, this would mean that a warning will be reported for the user and the field won't appear in the data.

This step is explained in more detail in Schema Inference

2. Adding types

When users and plugins add types using `createTypes`, those types are added to the schema composer. The types that don't have inference disabled will also

get types created from Schema Inference merged into them, with user created fields having priority. After that, inferred types that haven't been created are also added to the composer.

3. Legacy schema customization

Before schema customization was added, there were several ways that one could modify the schema. Those were the `createNodeField` action, `setFieldsOnGraphQLNodeType` API and `graphql-config.js` mappings.

`createNodeField`

This adds a field under the `fields` field. Plugins can't modify types that they haven't created, so you can use this method to add data to nodes that your plugin doesn't own. This doesn't modify the schema directly. Instead, those fields are picked by inference. There are no plans to deprecate this API at the moment.

`setFieldsOnGraphQLNodeType`

This allows adding GraphQL Fields to any node type. This operates on GraphQL types itself and the syntax matches `graphql-js` field definitions. This API will be marked as deprecated in Gatsby v3, moved under a flag in Gatsby v4, and removed from Gatsby v5. `createTypes` and `addResolvers` should solve all the use cases for this API.

`graphql-config.js` mapping

Node Type Mapping allows customizing schema by using site configuration. There are currently no plans to deprecate this API at the moment.

4. Parent / children relationships

Nodes can be connected into *child-parent* relationships either by using `createParentChildLink` or by adding the `parent` field to raw node data. Child types can always access parent with the `parent` field in GraphQL. Parent types also get `children` fields as well as "convenience child fields" `child[TypeName]` and `children[TypeName]`.

Children types are either inferred from data or created using `@childOf` directive, either by parent type name or by `mimeType` (only for File parent types).

5. Processing each type and adding root fields

See Schema Root Fields and Utility Types for a more detailed description of this step.

For each type, utility types are created. Those are input object types, used for searching, sorting and filtering, and types for paginated data (Connections and Edges).

For searching and sorting, Gatsby goes through every field in the type and converts them to corresponding Input GraphQL types. Scalars are converted to objects with filter operator fields like `eq` or `ni`. Object types are converted to Input Object types. For sorting, enums are created out of all fields so that one can use that to specify the sort.

Those types are used to create *root fields* of the schema in `Query` type. For each node type a root field for querying one item and paginated items are created (for example, for type `BlogPost` it would be `blogPost` and `allBlogPost`).

6. Merging in third-party schemas

If a plugin like `gatsby-source-graphql` is used, all third-party schemas that it provided are merged into the Gatsby schema.

7. Adding custom resolvers

`createResolvers` API is called, allowing users to add additional customization on top of created schema. This is an "escape hatch" API, as it allows to modify any fields or types in the Schema, including Query type.

8. Second schema build for `SitePage`

Because `SitePage` nodes are created after a schema is first created (at `createPages`) API call, the type of the `SitePage.context` field can change based on which context was passed to pages. Therefore, an additional schema inference pass happens and then the schema is updated.

Note that this behavior will be removed in Gatsby v3 and context will become a list of key/value pairs in GraphQL.