One of Gatsby's main strengths is the ability to query data from a variety of sources in a uniform way with GraphQL. For this to work, a GraphQL Schema must be generated that defines the shape of the data.

Gatsby is able to automatically infer a GraphQL Schema from your data, and in many cases, this is really all you need. There are however situations when you either want to explicitly define the data shape, or add custom functionality to the query layer - this is what Gatsby's Schema Customization API provides.

The following guide walks through some examples to showcase the API.

> *This guide is aimed at plugin authors, users trying to fix GraphQL schemas created by automatic type inference, developers optimizing builds for larger sites, and anyone interested in customizing Gatsby's schema generation. As such, the guide assumes that you're somewhat familiar with GraphQL types and with using Gatsby's Node APIs. For a higher level approach to using Gatsby with GraphQL, refer to the API reference.*

Explicitly defining data types

The example project is a blog that gets its data from local Markdown files which provide the post contents, as well as author information in JSON format. There are also occasional guest contributors whose info is kept in a separate JSON file.

```
---
title: Sample Post
publishedAt: 2019-04-01
author: jane@example.com
tags:
  - wow
---

# Heading

Text
```

```
[
  {
    "name": "Doe",
    "firstName": "Jane",
    "email": "jane@example.com",
    "joinedAt": "2018-01-01"
  }
]
```

```
[
  {
    "name": "Doe",
    "firstName": "Zoe",
    "email": "zoe@example.com",
    "receivedSwag": true
  }
]
```

To be able to query the contents of these files with GraphQL, they need to first be loaded into Gatsby's internal data store. This is what source and transformer plugin accomplish - in this case `gatsby-source-filesystem` and `gatsby-transformer-remark` plus `gatsby-transformer-json`. Every markdown post file is hereby transformed into a "node" object in the internal data store with a unique `id` and a type `MarkdownRemark`. Similarly, an author will be represented by a node object of type `AuthorJson`, and contributor info will be transformed into node objects of type `ContributorJson`.

The Node interface

This data structure is represented in Gatsby's GraphQL schema with the `Node` interface, which describes the set of fields common to node objects created by source and transformer plugins (`id`, `parent`, `children`, as well as a couple of `internal` fields like `type`). In GraphQL Schema Definition Language (SDL), it looks like this:

```
interface Node {
  id: ID!
  parent: Node!
  children: [Node!]!
  internal: Internal!
}

type Internal {
  type: String!
}
```

Types created by source and transformer plugins implement this interface. For example, the node type created by `gatsby-transformer-json` for `authors.json` will be represented in the GraphQL schema as:

```
type AuthorJson implements Node {
  id: ID!
  parent: Node!
  children: [Node!]!
  internal: Internal!
  name: String
  firstName: String
  email: String
  joinedAt: Date
}
```

> *A quick way to inspect the schema generated by Gatsby is the GraphQL Playground. Start your project with `GATSBY_GRAPHQL_IDE=playground gatsby develop`, open the playground at `http://localhost:8000/___graphql` and inspect the `Schema` tab on the right.*

Automatic type inference

It's important to note that the data in `author.json` does not provide type information of the Author fields by itself. In order to translate the data shape into GraphQL type definitions, Gatsby has to inspect the contents of every field and check its type. In many cases this works very well and it is still the default mechanism for creating a GraphQL schema.

There are however two problems with this approach: (1) it is quite time-consuming and therefore does not scale very well and (2) if the values on a field are of different types Gatsby cannot decide which one is the correct one. A consequence of this is that if your data sources change, type inference could suddenly fail.

Both problems can be solved by providing explicit type definitions for Gatsby's GraphQL schema.

Creating type definitions

Look at the latter case first. Assume a new author joins the team, but in the new author entry there is a typo on the `joinedAt` field: "201-04-02" which is not a valid Date.

```
+  {
+    "name": "Doe",
+    "firstName": "John",
+    "email": "john@example.com",
+    "joinedAt": "201-04-02"
+  }
]
```

This will confuse Gatsby's type inference since the `joinedAt` field will now have both Date and String values.

Fixing field types

To ensure that the field will always be of Date type, you can provide explicit type definitions to Gatsby with the `createTypes` action. It accepts type definitions in GraphQL Schema Definition Language:

```
exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions
  const typeDefs = `
    type AuthorJson implements Node {
      joinedAt: Date
    }
  `
  createTypes(typeDefs)
}
```

Note that the rest of the fields (`name` , `firstName` etc.) don't have to be provided, they will still be handled by Gatsby's type inference.

> *Actions to customize Gatsby's schema generation are made available in the `createSchemaCustomization` (available in Gatsby v2.12 and above), and `sourceNodes` APIs.*

Opting out of type inference

There are however advantages to providing full definitions for a node type, and bypassing the type inference mechanism altogether. With smaller scale projects inference is usually not a performance problem, but as projects grow the performance penalty of having to check each field type will become noticeable.

Gatsby allows to opt out of inference with the `@dontInfer` type directive - which in turn requires that you explicitly provide type definitions for all fields that should be available for querying:

```
exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions
  const typeDefs = `
    type AuthorJson implements Node @dontInfer {
      name: String!
      firstName: String!
      email: String!
      joinedAt: Date
    }
  `
  createTypes(typeDefs)
}
```

Note that you don't need to explicitly provide the Node interface fields (`id` , `parent` , etc.), Gatsby will automatically add them for you.

> *If you wonder about the exclamation marks - those allow specifying nullability in GraphQL, i.e. if a field value is allowed to be `null` or not.*

Defining media types

You can specify the media types handled by a node type using the `@mimeTypes` extension:

```
type Markdown implements Node
  @mimeTypes(types: ["text/markdown", "text/x-markdown"]) {
  id: ID!
}
```

The types passed in are used to determine child relations of the node.

Defining child relations

The `@childOf` extension can be used to explicitly define what node types or media types a node is a child of and immediately add `child[MyType]` and `children[MyType]` fields on the parent.

The `types` argument takes an array of strings and determines what node types the node is a child of:

```
# Adds `childMdx` as a field of `File` and `Markdown` nodes
type Mdx implements Node @childOf(types: ["File", "Markdown"]) {
  id: ID!
}
```

The `mimeTypes` argument takes an array of strings and determines what media types the node is a child of:

```
# Adds `childMdx` as a child of any node type with the `@mimeTypes` set to
"text/markdown" or "text/x-markdown"
type Mdx implements Node
  @childOf(mimeTypes: ["text/markdown", "text/x-markdown"]) {
  id: ID!
}
```

The `mimeTypes` and `types` arguments can be combined as follows:

```
# Adds `childMdx` as a child to `File` nodes *and* nodes with `@mimeTypes` set to
"text/markdown" or "text/x-markdown"
type Mdx implements Node
  @childOf(types: ["File"], mimeTypes: ["text/markdown", "text/x-markdown"]) {
  id: ID!
}
```

Nested types

So far, the example project has only been dealing with scalar values (`String` and `Date` ; GraphQL also knows `ID` , `Int` , `Float` , `Boolean` and `JSON`). Fields can however also contain complex object values. To target those fields in GraphQL SDL, you can provide a full type definition for the nested type, which can be arbitrarily named (as long as the name is unique in the schema). In the example project, the `frontmatter` field on the `MarkdownRemark` node type is a good example. Say you want to ensure that `frontmatter.tags` will always be an array of strings.

```
exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions
  const typeDefs = `
    type MarkdownRemark implements Node {
      frontmatter: Frontmatter
    }
    type Frontmatter {
      tags: [String!]!
    }
  `
  createTypes(typeDefs)
}
```

Note that with `createTypes` you cannot directly target a `Frontmatter` type without also specifying that this is the type of the `frontmatter` field on the `MarkdownRemark` type, The following would fail because Gatsby would have no way of knowing which field the `Frontmatter` type should be applied to:

```
exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions
  const typeDefs = `
    # This will fail!!!
    type Frontmatter {
      tags: [String]!
    }
  `
  createTypes(typeDefs)
}
```

It is useful to think about your data, and the corresponding GraphQL schema, by always starting from the Node types created by source and transformer plugins.

> *Note that the `Frontmatter` type must not implement the Node interface since it is not a top-level type created by source or transformer plugins: it has no `id` field, and is there to describe the data shape on a nested field.*

Gatsby Type Builders

In many cases, GraphQL SDL provides a succinct way to provide type definitions for your schema. If however you need more flexibility, `createTypes` also accepts type definitions provided with the help of Gatsby Type Builders, which are more flexible than SDL syntax but less verbose than `graphql-js`. They are accessible on the `schema` argument passed to Node APIs.

```
exports.createSchemaCustomization = ({ actions, schema }) => {
  const { createTypes } = actions
  const typeDefs = [
    schema.buildObjectType({
      name: "ContributorJson",
      fields: {
        name: "String!",
        firstName: "String!",
        email: "String!",
        receivedSwag: {
          type: "Boolean",
          resolve: source => source.receivedSwag || false,
        },
      },
      interfaces: ["Node"],
    }),
  ]
  createTypes(typeDefs)
}
```

Gatsby Type Builders allow referencing types as simple strings, and accept full field configs (`type` , `args` , `resolve`). When defining top-level types, don't forget to pass `interfaces: ['Node']` , which does the same for Type Builders as adding `implements Node` does for SDL-defined types. It is also possible to opt out of type inference with Type Builders by setting the `infer` type extension to `false` :

```
schema.buildObjectType({
  name: "ContributorJson",
  fields: {
    name: "String!",
  },
  interfaces: ["Node"],
+ extensions: {
+   // While in SDL you have two different directives, @infer and @dontInfer to
+   // control inference behavior, Gatsby Type Builders take a single `infer`
+   // extension which accepts a Boolean
+   infer: false
+ },
}),
```

> *Type Builders also exist for Input, Interface and Union types: `buildInputType`, `buildInterfaceType`, and `buildUnionType`. Note that the `createTypes` action also accepts `graphql-js` types directly, but usually either SDL or Type Builders are the better alternatives.*

Foreign-key fields

In the example project, the `frontmatter.author` field on `MarkdownRemark` nodes to expand the provided field value to a full `AuthorJson` node. For this to work, there has to be provided a custom field resolver. (see below for more info on `context.nodeModel`)

```
exports.createSchemaCustomization = ({ actions, schema }) => {
  const { createTypes } = actions
  const typeDefs = [
    "type MarkdownRemark implements Node { frontmatter: Frontmatter }",
    schema.buildObjectType({
      name: "Frontmatter",
      fields: {
        author: {
          type: "AuthorJson",
          resolve: (source, args, context, info) => {
            // If you were linking by ID, you could use `getNodeById` to
            // find the correct author:
            //
            // return context.nodeModel.getNodeById({
            //   id: source.author,
            //   type: "AuthorJson",
            // })
            //
            // But since the example is using the author email as foreign key,
            // you can use `nodeModel.findOne` to find the linked author node.
            // Note: Instead of getting all nodes and then using
Array.prototype.find()
            // Use nodeModel.findOne instead where possible!
            return context.nodeModel.findOne({
              type: "AuthorJson",
              query: {
                filter: { email: { eq: source.author } }
              }
            })
          },
        },
      },
    }),
  ]
  createTypes(typeDefs)
}
```

What is happening here is that you provide a custom field resolver that asks Gatsby's internal data store for the full node object with the specified `id` and `type`.

Because creating foreign-key relations is such a common use case, Gatsby luckily also provides a much easier way to do this -- with the help of extensions or directives. It looks like this:

```
type MarkdownRemark implements Node {
  frontmatter: Frontmatter
}
type Frontmatter {
  author: AuthorJson @link # default foreign-key relation by `id`
  reviewers: [AuthorJson] @link(by: "email") # foreign-key relation by custom field
}
type AuthorJson implements Node {
  posts: [MarkdownRemark] @link(by: "frontmatter.author.email", from: "email") #
easy back-ref
}
```

This example assumes that your markdown frontmatter is in the shape of:

```
---
reviewers:
  - jane@example.com
  - doe@example.com
---
```

And your author JSON looks like this:

```
[
  {
    "name": "Doe",
    "firstName": "Jane",
    "email": "jane@example.com"
  },
  {
    "name": "Doe",
    "firstName": "Zoe",
    "email": "zoe@example.com"
  }
]
```

You provide a `@link` directive on a field and Gatsby will internally add a resolver that is quite similar to the one written manually above. If no argument is provided, Gatsby will use the `id` field as the foreign-key, otherwise the foreign-key has to be provided with the `by` argument. The optional `from` argument allows getting the field on the current type which acts as the foreign-key to the field specified in `by`. In other words, you `link` **on** `from` **to** `by`. This makes `from` especially helpful when adding a field for back-linking.

For the above example you can read `@link` this way: Use the value from the field `Frontmatter.reviewers` and match it by the field `AuthorJson.email`.

Keep in mind that in the example above, the link of `posts` in `AuthorJson` works because `frontmatter` and `author` are both objects. If, for example, the `Frontmatter` type had a list of `authors` instead

(`frontmatter.authors.email`), it wouldn't work since the `by` argument doesn't support arrays. In that case, you'd have to provide a custom resolver with Gatsby Type Builders or createResolvers API.

Extensions and directives

Out of the box, Gatsby provides four extensions that allow adding custom functionality to fields without having to manually write field resolvers: the `link` extension has already been discussed above, `dateformat` allows adding date formatting options, `fileByRelativePath` is similar to `link` but will resolve relative paths when linking to `File` nodes, and `proxy` is helpful when dealing with data that contains field names with characters that are invalid in GraphQL.

To add an extension to a field you can either use a directive in SDL, or the `extensions` property when using Gatsby Type Builders:

```
exports.createSchemaCustomization = ({ actions, schema }) => {
  const { createTypes } = actions
  const typeDefs = [
    "type MarkdownRemark implements Node { frontmatter: Frontmatter }",
    `type Frontmatter {
      publishedAt: Date @dateformat(formatString: "DD-MM-YYYY")
    }`,
    schema.buildObjectType({
      name: "AuthorJson",
      fields: {
        joinedAt: {
          type: "Date",
          extensions: {
            dateformat: {},
          },
        },
      },
    }),
  ]
  createTypes(typeDefs)
}
```

The above example adds date formatting options to the `AuthorJson.joinedAt` and the `MarkdownRemark.frontmatter.publishedAt` fields. Those options are available as field arguments when querying those fields:

```
query {
  allAuthorJson {
    joinedAt(fromNow: true)
  }
}
```

`publishedAt` is also provided a default `formatString` which will be used when no explicit formatting options are provided in the query.

Setting default field values

For setting default field values, Gatsby currently does not (yet) provide an out-of-the-box extension, so resolving a field to a default value (instead of `null`) requires manually adding a field resolver. For example, to add a default tag to every blog post:

```
exports.createSchemaCustomization = ({ actions, schema }) => {
  const { createTypes } = actions
  const typeDefs = [
    "type MarkdownRemark implements Node { frontmatter: Frontmatter }",
    schema.buildObjectType({
      name: "Frontmatter",
      fields: {
        tags: {
          type: "[String!]",
          resolve(source, args, context, info) {
            // For a more generic solution, you could pick the field value from
            // `source[info.fieldName]`
            const { tags } = source
            if (source.tags == null || (Array.isArray(tags) && !tags.length)) {
              return ["uncategorized"]
            }
            return tags
          },
        },
      },
    }),
  ]
  createTypes(typeDefs)
}
```

Creating custom extensions

With the `createFieldExtension` action it is possible to define custom extensions as a way to add reusable functionality to fields. Say you want to add a `fullName` field to `AuthorJson` and `ContributorJson`.

You could write a `fullNameResolver`, and use it in two places:

```
const fullNameResolver = source => `${source.firstName} ${source.name}`

exports.createSchemaCustomization = ({ actions, schema }) => {
  actions.createTypes([
    {
      name: "AuthorJson",
      interfaces: ["Node"],
      fields: {
        fullName: {
          type: "String",
          resolve: fullNameResolver,
        },
      },
    },
    {
      name: "ContributorJson",
      interfaces: ["Node"],
      fields: {
        fullName: {
          type: "String",
          resolve: fullNameResolver,
        },
      },
    },
  ])
}
```

However, to make this functionality available to other plugins as well, and make it usable in SDL, you can register it as a field extension.

A field extension definition requires a name, and an `extend` function, which should return a (partial) field config (an object, with `type` , `args` , `resolve`) which will be merged into the existing field config.

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

  createFieldExtension({
    name: "fullName",
    extend(options, prevFieldConfig) {
      return {
        resolve(source) {
          return `${source.firstName} ${source.name}`
        },
      }
    },
  })

  createTypes(`
    type AuthorJson implements Node {
      fullName: String @fullName
    }
    type ContributorJson implements Node {
      fullName: String @fullName
    }
  `)
}
```

This approach becomes a lot more powerful when plugins provide custom field extensions. A *very* basic markdown transformer plugin could for example provide an extension to convert markdown strings into HTML:

```
const remark = require(`remark`)
const html = require(`remark-html`)

exports.createSchemaCustomization = ({ actions }) => {
  actions.createFieldExtension({
    name: "md",
    args: {
      sanitize: {
        type: "Boolean!",
        defaultValue: true,
      },
    },
    // The extension `args` (above) are passed to `extend` as
    // the first argument (`options` below)
    extend(options, prevFieldConfig) {
      return {
        args: {
          sanitize: "Boolean",
        },
        resolve(source, args, context, info) {
          const fieldValue = context.defaultFieldResolver(
            source,
            args,
            context,
            info
          )
          const shouldSanitize =
            args.sanitize != null ? args.sanitize : options.sanitize
          const processor = remark().use(html, { sanitize: shouldSanitize })
          return processor.processSync(fieldValue).contents
        },
      }
    },
  })
}
```

It can then be used in any `createTypes` call by adding the directive/extension to the field:

```
exports.createSchemaCustomization = ({ actions }) => {
  actions.createTypes(`
    type BlogPost implements Node {
      content: String @md
    }
  `)
}
```

Note that in the above example, there have been additional provided configuration options with `args`. This is e.g. useful to provide default field arguments:

```
exports.createSchemaCustomization = ({ actions }) => {
  actions.createTypes(`
    type BlogPost implements Node {
      content: String @md(sanitize: false)
    }
  `)
}
```

Also note that field extensions can decide themselves if an existing field resolver should be wrapped or overwritten. The above examples have all decided to return a new `resolve` function. Because the `extend` function receives the current field config as its second argument, an extension can also decide to wrap an existing resolver:

```
extend(options, prevFieldConfig) {
+  const { resolve } = prevFieldConfig
+  return {
+    async resolve(source, args, context, info) {
+      const resultFromPrevResolver = await resolve(source, args, context, info)
      /* ... */
+      return processor.processSync(resultFromPrevResolver).contents
+    }
+  }
}
```

If multiple field extensions are added to a field, resolvers are processed in this order: first a custom resolver added with `createTypes` (or `createResolvers`) runs, then field extension resolvers execute from left to right.

Finally, note that in order to get the current `fieldValue` , you use `context.defaultFieldResolver` .

createResolvers API

While it is possible to directly pass `args` and `resolvers` along the type definitions using Gatsby Type Builders, an alternative approach specifically tailored towards adding custom resolvers to fields is the `createResolvers` Node API.

```
exports.createResolvers = ({ createResolvers }) => {
  const resolvers = {
    Frontmatter: {
      author: {
        type: "AuthorJson",
        resolve(source, args, context, info) {
          return context.nodeModel.getNodeById({
            id: source.author,
            type: "AuthorJson",
          })
        },
      },
    },
  }
  createResolvers(resolvers)
}
```

Note that `createResolvers` allows adding new fields to types, modifying `args` and `resolver` -- but not overriding the field type. This is because `createResolvers` is run last in schema generation, and modifying a field type would mean having to regenerate corresponding input types (`filter` , `sort`), which you want to avoid. If possible, specifying field types should be done with the `createTypes` action.

Accessing Gatsby's data store from field resolvers

As mentioned above, Gatsby's internal data store and query capabilities are available to custom field resolvers on the `context.nodeModel` argument passed to every resolver. Accessing node(s) by `id` (and optional `type`) is possible with [`getNodeById`] and [`getNodesByIds`] . To get all nodes, or all nodes of a certain type, use [`findAll`] . And running a query from inside your resolver functions can be accomplished with `findAll` as well, which accepts `filter` and `sort` query arguments.

You could for example add a field to the `AuthorJson` type that lists all recent posts by an author:

```
exports.createResolvers = ({ createResolvers }) => {
  const resolvers = {
    AuthorJson: {
      recentPosts: {
        type: ["MarkdownRemark"],
        resolve: async (source, args, context, info) => {
          const { entries } = await context.nodeModel.findAll({
            query: {
              filter: {
                frontmatter: {
                  author: { eq: source.email },
                  date: { gt: "2019-01-01" },
                },
              },
            },
            type: "MarkdownRemark",
          })
          return entries
        },
      },
    },
  }
  createResolvers(resolvers)
}
```

When using `findAll` to sort query results, be aware that both `sort.fields` and `sort.order` are `GraphQLList` fields. Also, nested fields on `sort.fields` have to be provided in dot-notation (not separated by triple underscores). For example:

```
context.nodeModel.findAll({
  query: {
    sort: {
      fields: ["frontmatter.publishedAt"],
      order: ["DESC"],
    },
  },
  type: "MarkdownRemark",
})
```

Custom query fields

One powerful approach enabled by `createResolvers` is adding custom root query fields. While the default root query fields added by Gatsby (e.g. `markdownRemark` and `allMarkdownRemark`) provide the whole range of query options, query fields designed specifically for your project can be useful. For example, you can add a query field for all external contributors to the example blog who have received their swag:

```
exports.createResolvers = ({ createResolvers }) => {
  const resolvers = {
    Query: {
      contributorsWithSwag: {
        type: ["ContributorJson"],
        resolve: async (source, args, context, info) => {
          const { entries } = await context.nodeModel.findAll({
            query: {
              filter: {
                receivedSwag: { eq: true },
              },
            },
            type: "ContributorJson",
          })

          return entries
        },
      },
    },
  }
  createResolvers(resolvers)
}
```

Because you might also be interested in the reverse - which contributors haven't received their swag yet - why not add a (required) custom query arg?

```
exports.createResolvers = ({ createResolvers }) => {
  const resolvers = {
    Query: {
      contributors: {
        type: ["ContributorJson"],
        args: {
          receivedSwag: "Boolean!",
        },
        resolve: async (source, args, context, info) => {
          const { entries } = await context.nodeModel.findAll({
            query: {
              filter: {
                receivedSwag: { eq: args.receivedSwag },
              },
            },
            type: "ContributorJson",
          })

          return entries
        },
      },
    },
  }
  createResolvers(resolvers)
}
```

It is also possible to provide more complex custom input types which can be defined directly inline in SDL. You could for example add a field to the `ContributorJson` type that counts the number of posts by a contributor, and then add a custom root query field `contributors` which accepts `min` or `max` arguments to only return contributors who have written at least `min`, or at most `max` number of posts:

```
exports.createResolvers = ({ createResolvers }) => {
  const resolvers = {
    Query: {
      contributors: {
        type: ["ContributorJson"],
        args: {
          postsCount: "input PostsCountInput { min: Int, max: Int }",
        },
        resolve: async (source, args, context, info) => {
          const { max, min = 0 } = args.postsCount || {}
          const operator = max != null ? { lte: max } : { gte: min }

          const { entries } = await context.nodeModel.findAll({
            query: {
              filter: {
                posts: operator,
              },
            },
            type: "ContributorJson",
          })

          return entries
        },
      },
    },
    ContributorJson: {
      posts: {
        type: `Int`,
        resolve: async (source, args, context, info) => {
          const { entries } = await context.nodeModel.findAll({ type:
"MarkdownRemark" })
          const posts = entries.filter(post => post.frontmatter.author ===
source.email)
          return Array.from(posts).length
        },
      },
    },
  }
  createResolvers(resolvers)
}
```

Taking care of hot reloading

When creating custom field resolvers, it is important to ensure that Gatsby knows about the data a page depends on for hot reloading to work properly. When you retrieve nodes from the store with `context.nodeModel` methods, it is usually not necessary to do anything manually, because Gatsby will register dependencies for the query results automatically. If you want to customize this, you can add a page data dependency either programmatically with `context.nodeModel.trackPageDependencies` , or with:

```
context.nodeModel.findAll(
  { type: "MarkdownRemark" },
  { connectionType: "MarkdownRemark" }
)
```

Custom Interfaces and Unions

Finally, say you want to have a page on the example blog that lists all team members (authors and contributors). What you could do is have two queries, one for `allAuthorJson` and one for `allContributorJson` and manually merge those. GraphQL however provides a more elegant solution to these kinds of problems with "abstract types" (Interfaces and Unions). Since authors and contributors actually share most of the fields, you can abstract those up into a `TeamMember` interface and add a custom query field for all team members (as well as a custom resolver for full names):

```
exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions
  const typeDefs = `
    interface TeamMember {
      name: String!
      firstName: String!
      email: String!
    }

    type AuthorJson implements Node & TeamMember {
      name: String!
      firstName: String!
      email: String!
      joinedAt: Date
    }

    type ContributorJson implements Node & TeamMember {
      name: String!
      firstName: String!
      email: String!
      receivedSwag: Boolean
    }
  `
  createTypes(typeDefs)
}

exports.createResolvers = ({ createResolvers }) => {
  const fullName = {
    type: "String",
    resolve(source, args, context, info) {
      return source.firstName + " " + source.name
    },
  }
  const resolvers = {
    Query: {
      allTeamMembers: {
        type: ["TeamMember"],
        resolve: async (source, args, context, info) => {
          // Whenever possible, use `limit` and `skip` on findAll calls to increase
performance
          const { entries } = await context.nodeModel.findAll({ type: "TeamMember",
query: { limit: args.limit, skip: args.skip } })

          return entries
        },
      },
    },
    AuthorJson: {
      fullName,
    },
    ContributorJson: {
      fullName,
    },
  }
  createResolvers(resolvers)
}
```

To use the newly added root query field in a page query to get the full names of all team members, you could write:

```
export const query = graphql`
  {
    allTeamMembers {
      ... on AuthorJson {
        fullName
      }
      ... on ContributorJson {
        fullName
      }
    }
  }
`
```

Queryable interfaces

Since Gatsby 3.0.0, you can use interface inheritance to achieve the same thing as above: `TeamMember implements Node`. This will treat the interface like a normal top-level type that implements the `Node` interface, and thus automatically add root query fields for the interface.

```
exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions
  const typeDefs = `
    interface TeamMember implements Node {
      id: ID!
      name: String!
      firstName: String!
      email: String!
    }

    type AuthorJson implements Node & TeamMember {
      name: String!
      firstName: String!
      email: String!
      joinedAt: Date
    }

    type ContributorJson implements Node & TeamMember {
      name: String!
      firstName: String!
      email: String!
      receivedSwag: Boolean
    }
  `
  createTypes(typeDefs)
}
```

When querying, use inline fragments for the fields that are specific to the types implementing the interface (i.e. fields that are not shared):

```
export const query = graphql`
  {
    allTeamMember {
      nodes {
        name
        firstName
        email
        __typeName
        ... on AuthorJson {
          joinedAt
        }
        ... on ContributorJson {
          receivedSwag
        }
        ... on Node {
          parent {
            id
          }
        }
      }
    }
  }
`
```

Including the `__typeName` introspection field allows to check the node type when iterating over the query results in your component:

```
data.allTeamMember.nodes.map(node => {
  switch (node.__typeName) {
    case `AuthorJson`:
      return <Author {...node} />
    case `ContributorJson`:
      return <Contributor {...node} />
  }
})
```

> *Note: All types implementing a queryable interface must also implement the `Node` interface*