Upgrade Your Source Plugins for Gatsby 4

import { Announcement } from “gatsby-interface”

Gatsby 4 is here! Following on the heels of Gatsby 3, Gatsby 4 further improves build performance and introduces new parallel processing capabilities. In the guide below, we’ll walk you through preparing your source plugin for Gatsby 4. You’ll find this guide useful if you are a maintainer for a source plugin (as opposed to a consumer using a source plugin in your Gatsby site).

Introducing support for Gatsby 4 in your source plugin can be accomplished by ensuring your code adopts the 4 following changes. Many plugins already had the majority of their code organized the way it needed to be!

With Gatsby 4, Core APIs are being split into different processes so they’re able to run simultaneously in parallel instead of sequentially. Early results show at least 40% improvement of build performance. Gatsby 4 also lays the groundwork for two new options of rendering your Gatsby site: Server Side Rendering (SSR) and Deferred Static Generation (DSG) to scale Gatsby to infinity and beyond.

It’s time to get into it! The rest of this guide outlines the breaking changes in Gatsby 4 and some quick ways to resolve them. Find something confusing? Let us know in the GitHub discussion and we’ll respond as fast as possible.

<Announcement style={{marginBottom: “1.5rem”}}>

Looking for examples of source plugins that support Gatsby 4? Check out `gatsby-source-wordpress` and `gatsby-source-shopify`.

1. Modification to Gatsby’s GraphQL schema during `sourceNodes` is not allowed

There are three APIs that can modify Gatsby’s GraphQL schema: `createTypes`, `addThirdPartySchema`, `createFieldExtension`. In Gatsby 4, you’re no longer allowed to use these APIs during the `sourceNodes` lifecycle. Please use them in the `createSchemaCustomization` lifecycle instead.

The Old Way

`createTypes` is used inside `sourceNodes`.

```
exports.sourceNodes = ({ actions }) => { // highlight-line
  const { createTypes } = actions;

  createTypes(`
    type AuthorJson implements Node {
      joinedAt: Date
    }
  `)
}
```

The New Way

`createTypes` is used instead in `createSchemaCustomization`

```
exports.createSchemaCustomization = ({ actions }) => { // highlight-line
  const { createTypes } = actions;

  createTypes(`
    type AuthorJson implements Node {
      joinedAt: Date
    }
  `)
}
```

2. Data mutations need to happen during `sourceNodes` or `onCreateNode`

Creation or augmentation of nodes needs to happen in the appropriate APIs. In Gatsby 4, creating nodes inside resolvers is not allowed. You need to create them in `sourceNodes` and add expensive operations inside your resolvers.

The Old Way

A node is created at the same time a resolver is created.

```
exports.createResolvers = ({ // highlight-line
  createNodeId,
  actions,
  createResolvers,
  store,
  cache,
  reporter,
}) => {
  createResolvers({
    CustomImage: {
      localImage: {
        type: "File!",
```

```
        resolve: async (source, args, context, info) => {
          return createRemoteFileNode({ // highlight-line
            url: source.url,
            parentNodeId: source.id,
            store,
            cache,
            createNode: actions.createNode,
            createNodeId,
            reporter,
          })
        },
      },
    },
  })
}
```

The New Way

The type is created in `createSchemaCustomization` and then referenced inside `sourceNodes` to create the node.

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

  createTypes(`
    type CustomImage implements Node {
      localImage: File @link
    }
  `)
}

exports.sourceNodes = async ({ // highlight-line
  actions,
  createNodeId,
  createContentDigest,
  store,
  cache,
  reporter,
}) => {
  const { createNode } = actions

  // code to fetch data

  for (const { url } of remoteImages) {
    const nodeId = createNodeId(`my-data-${url}`)
    const image = await createRemoteFileNode({ // highlight-line
```

```
      url: url,
      parentNodeId: nodeId,
      store,
      cache,
      createNode,
      createNodeId,
      reporter,
    })
    const node = {
      id: nodeId,
      parent: null,
      children: [],
      url,
      localImageId: image.id,
      internal: {
        type: `CustomImage`,
        content: url,
        contentDigest: createContentDigest(url),
      },
    }

    createNode(node)
  }
}
```

3. Global state

As stated above, Gatsby 4 improves build times by running GraphQL queries in parallel. Running multiple processes means global variables might not contain the data that you hoped they would.

When a new process (worker) gets created, the new `onPluginInit` lifecycle method is run, which can be used to restore global state if needed.

The Old Way

`onPreBootstrap` (or `onPreInit`) is setting global state so that other functions can access that global state.

```
let globalPluginOptions = {}

exports.onPreBootstrap = (_, pluginOptions) => { // highlight-line
  globalPluginOptions = pluginOptions
}

function aDeepNestedFunction(arg) {
  if (globalPluginOptions.convert) {
```

```
    return arg.toUpperCase()
  } else {
    return arg
  }
}
```

The New Way

To check if the new `onPluginInit` lifecycle is available, you can use the `isGatsbyNodeLifecycleSupported` from the `gatsby-plugin-utils` package. (Make sure to add this explicitly to your dependencies by running `npm install gatsby-plugin-utils`!) This will help you keep backwards compatibility with Gatsby 3 while moving forward to a Gatsby 4 world. You'll also need to check if you are using the `stable` or `unstable` version of `onPluginInit`.

```
let coreSupportsOnPluginInit: "unstable" | "stable" | undefined

try {
  const { isGatsbyNodeLifecycleSupported } = require(`gatsby-plugin-utils`)
  if (isGatsbyNodeLifecycleSupported(`onPluginInit`)) {
    coreSupportsOnPluginInit = "stable"
  } else if (isGatsbyNodeLifecycleSupported(`unstable_onPluginInit`)) {
    coreSupportsOnPluginInit = "unstable"
  }
} catch (e) {
  console.error(`Could not check if Gatsby supports onPluginInit lifecycle`)
}

let globalPluginOptions = {}

const initializeGlobalState = (_, pluginOptions) => {
  globalPluginOptions = pluginOptions
}

if (coreSupportsOnPluginInit === "stable") {
  exports.onPluginInit = initializeGlobalState // highlight-line
} else if (coreSupportsOnPluginInit === "unstable") {
  exports.unstable_onPluginInit = initializeGlobalState // highlight-line
} else {
  exports.onPreBootstrap = initializeGlobalState // highlight-line
}

function aDeepNestedFunction(arg) {
  if (globalPluginOptions.convert) {
    return arg.toUpperCase()
  } else {
```

```
    return arg
  }
}
```

4. Custom error map needs to be initiated in `onPluginInit`

To make it more clear for users when things go wrong, Gatsby has structured logs. It allows plugins to properly guide people to a solution by providing metadata such as URLs to docs. This feature needs to be initialized in `onPluginInit` to allow GraphQL resolvers to report these errors.

The Old Way

`onPreInit` is setting the error map.

```
const ERROR_MAP = {
  [CODES.Generic]: {
    text: context => context.sourceMessage,
    level: `ERROR`,
    type: `PLUGIN`,
  },
  [CODES.MissingResource]: {
    text: context => context.sourceMessage,
    level: `ERROR`,
    type: `PLUGIN`,
    category: `USER`,
  },
}
exports.onPreInit = ({ reporter }) => { // highlight-line
  if (reporter.setErrorMap) {
    reporter.setErrorMap(ERROR_MAP)
  }
}
```

The New Way

Similar to handling global state, you can use the `isGatsbyNodeLifecycleSupported` helper function from `gatsby-plugin-utils` to check if the new `onPluginInit` lifecycle method is available. (Make sure to add this explicitly to your dependencies by running `npm install gatsby-plugin-utils`!) This will help you keep backwards compatibility with Gatsby 3 while moving forward to a Gatsby 4 world. You'll also need to check if you are using the `stable` or `unstable` version of `onPluginInit`.

```
let coreSupportsOnPluginInit: "unstable" | "stable" | undefined

try {
```

```
  const { isGatsbyNodeLifecycleSupported } = require(`gatsby-plugin-utils`)
  if (isGatsbyNodeLifecycleSupported(`onPluginInit`)) {
    coreSupportsOnPluginInit = "stable"
  } else if (isGatsbyNodeLifecycleSupported(`unstable_onPluginInit`)) {
    coreSupportsOnPluginInit = "unstable"
  }
} catch (e) {
  console.error(`Could not check if Gatsby supports onPluginInit lifecycle`)
}

const ERROR_MAP = {
  [CODES.Generic]: {
    text: context => context.sourceMessage,
    level: `ERROR`,
    type: `PLUGIN`,
  },
  [CODES.MissingResource]: {
    text: context => context.sourceMessage,
    level: `ERROR`,
    type: `PLUGIN`,
    category: `USER`,
  },
}

const initializePlugin = ({ reporter }) => {
  if (reporter.setErrorMap) {
    reporter.setErrorMap(ERROR_MAP)
  }
}

// need to conditionally export otherwise it throws an error for older versions
if (coreSupportsOnPluginInit === "stable") {
  exports.onPluginInit = initializePlugin // highlight-line
} else if (coreSupportsOnPluginInit === "unstable") {
  exports.unstable_onPluginInit = initializePlugin // highlight-line
} else {
  exports.onPreInit = initializePlugin // highlight-line
}
```

5. Bundling External Files

In order for DSG & SSR to work Gatsby creates bundles with all the contents of the site, plugins, and data. When a plugin (or your own `gatsby-node.js`) requires an external file via `fs` module (e.g. `fs.readFile`) the engine won't be able to include the file. As a result you might see an error (when trying to run DSG) like `ENOENT: no such file or directory` in the CLI.

This limitation applies to these lifecycle APIs: `setFieldsOnGraphQLNodeType`, `createSchemaCustomization`, and `createResolvers`.

Instead you should move the contents to a JS/TS file and import the file as this way the bundler will be able to include the contents.

The Old Way

Previously you might have required a `.gql` file to use it in one of Gatsby's APIs with the `fs` module:

```
const fs = require("fs")
const path = require("path")

exports.createSchemaCustomization = ({ actions }) => {
  const { createTypes } = actions
  const typeDefs = fs.readFileSync(
    // .gql file with the SDL
    path.resolve(__dirname, "schema.gql"),
    "utf8"
  )

  createTypes(typeDefs)
}
```

The New Way

You can either move the definitions to a JS/TS file or inline it in `createTypes` directly.

```
// JS file containing the SDL strings now
const typeDefs = require("./schema")

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

  createTypes(typeDefs)
}
```

Recommendations for publishing your new source plugin version

Publish a new version of your Gatsby-4-compatible package that references `"gatsby": "^4.0.0"` in its `peerDependencies` to signal that the given source plugin version is specifically updated to work with Gatsby 4.