Gatsby makes it easy to programmatically control your pages.

Pages can be created in three ways:

- By creating React components in the `src/pages` directory. (Note that you must make the component the default export.)
- By using the File System Route API to programmatically create pages from GraphQL and to create client-only routes.
- In your site's `gatsby-node.js` file, by implementing the API `createPages`. (Plugins can also implement `createPages` and create pages for you.)

Pages can also be modified by you after their creation. For example, you could change the `path` to create internationalized routes (see gatsby-theme-i18n for instance) by implementing the API `onCreatePage`.

> ***Note:*** *For most use cases you'll be able to use the File System Route API to create pages. Please read on if you need more control over the page creation or consume data outside of Gatsby's GraphQL data layer.*

Debugging help

To see what pages are being created by your code or plugins, you can query for page information while developing in Graph*i*QL. Paste the following query in the Graph*i*QL IDE for your site. The Graph*i*QL IDE is available when running your site's development server at `HOST:PORT/___graphql` e.g. `http://localhost:8000/___graphql`.

```
{
  allSitePage {
    edges {
      node {
        path
        component
        pluginCreator {
          name
          pluginFilepath
        }
      }
    }
  }
}
```

The `context` property accepts an object, and you can pass in any data you want the page to be able to access.

You can also query for any `context` data you or plugins added to pages.

> ***NOTE:*** *There are a few reserved names that cannot be used in `context`. They are: `path`, `matchPath`, `component`, `componentChunkName`, `pluginCreator___NODE`, and `pluginCreatorId`.*

Creating pages in gatsby-node.js

Often you will need to programmatically create pages. For example, you have markdown files where each should be a page.

This example assumes that each markdown page has a `path` set in the frontmatter of the markdown file.

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

// Implement the Gatsby API “createPages”. This is called once the
// data layer is bootstrapped to let plugins create pages from data.
exports.createPages = async ({ graphql, actions, reporter }) => {
  const { createPage } = actions

  // Query for markdown nodes to use in creating pages.
  const result = await graphql(
    `
      {
        allMarkdownRemark(limit: 1000) {
          edges {
            node {
              frontmatter {
                path
              }
            }
          }
        }
      }
    `
  )

  // Handle errors
  if (result.errors) {
    reporter.panicOnBuild(`Error while running GraphQL query.`)
    return
  }

  // Create pages for each markdown file.
  const blogPostTemplate = path.resolve(`src/templates/blog-post.js`)
  result.data.allMarkdownRemark.edges.forEach(({ node }) => {
    const path = node.frontmatter.path
    createPage({
      path,
      component: blogPostTemplate,
      // In your blog post template's graphql query, you can use pagePath
      // as a GraphQL variable to query for data from the markdown file.
      context: {
        pagePath: path,
      },
    })
  })
}
```

Trade-offs of querying for all fields in the context object of `gatsby-node.js`

Imagine a scenario where you could query for all the parameters your template would need in the `gatsby-node.js`. What would the implications be? In this section, you will look into this.

In the initial approach you have seen how the `gatsby-node.js` file would have a query block like so :

```
  const queryResults = await graphql(`
    query AllProducts {
      allProducts {
        nodes {
          id
        }
      }
    }
  `);
```

Using the `id` as an access point to query for other properties in the template is the default approach. However, suppose you had a list of products with properties you would like to query for. Handling the query entirely from `gatsby-node.js` would result in the query looking like this:

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

exports.createPages = async ({ graphql, actions }) => {
  const { createPage } = actions
  const queryResults = await graphql(`
    query AllProducts {
      allProducts {
        nodes {
          id
          name
          price
          description
        }
      }
    }
  `)

  const productTemplate = path.resolve(`src/templates/product.js`)
  queryResults.data.allProducts.nodes.forEach(node => {
    createPage({
      path: `/products/${node.id}`,
      component: productTemplate,
      context: {
        // This time the entire product is passed down as context
        product: node,
      },
    })
  })
}
```

> *You are now requesting all the data you need in a single query (this requires server-side support to fetch many products in a single database query).*

> *As long as you can pass this data down to the template component via `pageContext`, there is no need for the template to make a GraphQL query at all.*

Your template file would look like this:

```
function Product({ pageContext }) {
  const { product } = pageContext
  return (
    <div>
      Name: {product.name}
      Price: {product.price}
      Description: {product.description}
    </div>
  )
}
```

Performance implications

Using the `pageContext` props in the template component can come with its performance advantages, of getting in all the data you need at build time; from the createPages API. This removes the need to have a GraphQL query in the template component.

It does come with the advantage of querying your data from one place after declaring the `context` parameter.

However, it doesn't give you the opportunity to know what exactly you are querying for in the template and if any changes occur in the component query structure in `gatsby-node.js`. Hot reload is taken off the table and the site needs to be rebuilt for changes to reflect.

Gatsby stores page metadata (including context) in a Redux store (which also means that it stores the memory of the page). For larger sites (either number of pages and/or amount of data that is being passed via page context) this will cause problems. There might be "out of memory" crashes if it's too much data or degraded performance.

> *If there is memory pressure, Node.js will try to garbage collect more often, which is a known performance issue.*

Page query results are not stored in memory permanently and are being saved to disk immediately after running the query.

We recommend passing "ids" or "slugs" and making full queries in the page template query to avoid this.

Incremental builds trade-off of this method

Another disadvantage of querying all of your data in `gatsby-node.js` is that your site has to be rebuilt every time you make a change, so you will not be able to take advantage of incremental builds.

Modifying pages created by core or plugins

Gatsby core and plugins can automatically create pages for you. Sometimes the default isn't quite what you want and you need to modify the created page objects.

Removing trailing slashes

A common reason for needing to modify automatically created pages is to remove trailing slashes.

To do this, in your site's `gatsby-node.js` add code similar to the following:

Note: There's also a plugin that will remove all trailing slashes from pages automatically: [gatsby-plugin-remove-trailing-slashes](#).

Note: If you need to perform an asynchronous action within `onCreatePage` you can return a promise or use an `async` function.

```
// Replacing '/' would result in empty string which is invalid
const replacePath = path => (path === `/` ? path : path.replace(/\/$/, ``))
// Implement the Gatsby API “onCreatePage”. This is
// called after every page is created.
exports.onCreatePage = ({ page, actions }) => {
  const { createPage, deletePage } = actions

  const oldPage = Object.assign({}, page)
  // Remove trailing slash unless page is /
  page.path = replacePath(page.path)
  if (page.path !== oldPage.path) {
    // Replace old page with new page
    deletePage(oldPage)
    createPage(page)
  }
}
```

Pass context to pages

The automatically created pages can receive context and use that as variables in their GraphQL queries. To override the default and pass your own context, open your site's `gatsby-node.js` and add similar to the following:

```
exports.onCreatePage = ({ page, actions }) => {
  const { createPage, deletePage } = actions

  deletePage(page)
  // You can access the variable "house" in your page queries now
  createPage({
    ...page,
    context: {
      ...page.context,
      house: `Gryffindor`,
    },
  })
}
```

On your pages and templates, you can access your context via the prop `pageContext` like this:

```
import React from "react"

const Page = ({ pageContext }) => {
```

```
  return <div>{pageContext.house}</div>
}

export default Page
```

Page context is serialized before being passed to pages. This means it can't be used to pass functions into components and `Date` objects will be serialized into strings.

Optimizing pages for Content Sync

When using the Content Sync feature on Gatsby Cloud, an optional parameter, `ownerNodeId`, can be passed to the `createPage` action to allow greater control over where content is previewed. By passing a value to `ownerNodeId`, you can ensure that Content Sync will redirect content authors to the page they intend to preview their content on. The value of `ownerNodeId` should be set to the id of the node that's the preferred node to preview for each page. This is typically the id of the node that's used to create the page path for each page.

```
const posts = result.data.allPosts.nodes

posts.forEach((post) => {
  createPage({
    path: `/blog/${post.slug}/`,
    component: blogPost,
    context: {},
    ownerNodeId: post.id, // highlight-line
  })
})
```

Creating client-only routes

In specific cases, you might want to create a site with client-only portions that are gated by authentication. For more on how to achieve this, refer to client-only routes & user authentication.