As methodologies for building Gatsby Themes begin to formalize and standardize, documentation will be added here. These aren't intended to be the only way to solve things, but are recommended approaches. If you have other ideas and best practices please open up a PR to update this page.

Naming

It's required to prefix themes with `gatsby-theme-`. So if you'd like to name your theme "awesome" you can name it `gatsby-theme-awesome` and place that as the `name` key in your `package.json`. Prefixing themes with `gatsby-theme-` enables Gatsby in identifying theme packages for compilation.

Initializing required directories

If your theme relies on the presence of particular directories, like `posts` for `gatsby-source-filesystem`, you can use the `onPreBootstrap` hook to initialize them to avoid a crash when Gatsby tries to build the site.

```
exports.onPreBootstrap = ({ store, reporter }) => {
  const { program } = store.getState()

  const dirs = [
    path.join(program.directory, "posts"),
    path.join(program.directory, "src/pages"),
    path.join(program.directory, "src/data"),
  ]

  dirs.forEach(dir => {
    if (!fs.existsSync(dir)) {
      reporter.log(`creating the ${dir} directory`)
      mkdirp.sync(dir)
    }
  })
}
```

Separating queries and presentational components

As a theme author, it's preferable to separate your data gathering and the components that render the data. This makes it easier for end users to be able to shadow a component like `PostList` or `AuthorCard` without having to write a pageQuery or StaticQuery.

Page queries

You can use a template for top-level data collection with a page query that passes the data to a `PostList` component:

```
import React from "react"
import { graphql } from "gatsby"

import PostList from "../components/PostList"

export default function MyPostsList(props) {
  return <PostList posts={props.allMdx.edges} />
}

export const query = graphql`
  query {
    allMdx(
      sort: { order: DESC, fields: [frontmatter___date] }
      filter: { frontmatter: { draft: { ne: true } } }
    ) {
      edges {
        node {
          id
          parent {
            ... on File {
              name
              sourceInstanceName
            }
          }
          frontmatter {
            title
            path
            date(formatString: "MMMM DD, YYYY")
          }
        }
      }
    }
  }
`
```

Static queries

You can use static queries at the top level template as well and pass the data to other presentational components as props:

```
import React from "react"
import { useStaticQuery, graphql } from "gatsby"

import Header from "../header.js"
import Footer from "../footer.js"

const Layout = ({ children }) => {
  const {
    site: { siteMetadata },
  } = useStaticQuery(
    graphql`
      query {
        site {
          siteMetadata {
            title
            social {
              twitter
              github
          }
        }
      }
    }
  `
  )

  const { title, social } = siteMetadata

  return (
    <>
      <Header title={title} />
      <main>{children}</main>
      <Footer {...social} />
    </>
  )
}

export default Layout
```

Site metadata

For commonly customized things, such as site title and social media handles, you can have the user set site metadata in their `gatsby-config.js`. Then, throughout your theme you can create a StaticQuery to access it:

```
import { graphql, useStaticQuery } from "gatsby"

export default function useSiteMetadata() {
  const data = useStaticQuery(graphql`
    {
      site {
        siteMetadata {
          title
          social {
            twitter
            github
            instagram
          }
        }
      }
    }
  `)

  return data.site.siteMetadata
}
```

Then use it in components like a header:

```
import React from "react"
import { Link } from "gatsby"
import useSiteMetadata from "../hooks/use-site-metadata"

export default function Header() {
  const { title, social } = useSiteMetadata()

  return (
    <header>
      <Link to="/">{title}</Link>
      <nav>
        <a href={`https://twitter.com/${social.twitter}`}>Twitter</a>
        <a href={`https://github.com/${social.github}`}>GitHub</a>
        <a href={`https://instagram.com/${social.instagram}`}>Instagram</a>
      </nav>
    </header>
  )
}
```

Breaking changes

Since themes will typically be installed by an end user from npm, it's important to follow semantic versioning, commonly referred to as semver.

This will allow your users to quickly discern how a dependency update might affect them. Patches and minor versions are not considered breaking changes, major versions are.

Patch *(0.0.X)*

Patches are defined as bug fixes that are done in a backwards-compatible way. This means that public facing APIs are unaffected.

Examples of patch versions

- **Fixing a bug** in a component, such as fixing a warning or adding a fallback value.
- **Upgrading dependencies** to their latest minor and patch versions.

Minor *(0.X.0)*

Minor versions are defined as new features or functionality that are added in a backwards-compatible way. This means that *existing* public facing APIs are unaffected.

Examples of minor versions

- **Adding new pages or queries** to your theme. For example, adding tag pages to a blog.
- **Adding new configuration options** to further customize a theme.
- **Displaying additional data** such as displaying excerpts to a post list.
- **Adding additional props to a component** for new functionality.
- **Adding a new MDX shortcode** that users can opt into.

Major *(X.0.0)*

Major versions are any bugfixes or new features that have been added without full backwards-compatibility. These are often called "breaking changes".

These changes should be accompanied by a migration guide that users can follow along for performing a theme upgrade.

Examples of major versions

- **Changing a filename in `src`** will always be a breaking change due to shadowing.
    - Moving where a query occurs
    - Renaming a component
    - Renaming a directory
- **Removing or changing the props a component accepts** since it will affect component extending.
- **Changing queries** since a user could be using the original data in shadowed components.
- **Removing or changing the behavior** of your theme's configuration.
- **Removing attributes in schema definitions** because it can break end user queries.
- **Removing default data** this could change your generated schema and break a user's site if they depend on some part of that generated schema.
- **Changing plugins or plugin configuration** such as removing a remark plugin as it will change the behavior of MD/MDX rendering.