Move source directories (pages, components, utils, etc.) under `src`

We moved site source files under a "src" directory to cleanly separate them from config/data/build folders so to make integration with various JavaScript tooling (e.g. Prettier) simpler.

Everything related to webpack, loaders, Babel, React should work nearly identically under v1 of Gatsby compared to v0.

```
mkdir src
git mv pages src
git mv components src
git mv utils src
...
```

Replace react-router's Link component with gatsby-link

`gatsby-link` is a wrapper for the `<Link>` component in react-router. It automatically prefixes urls and handles prefetching. Add `gatsby-link` to your project by running:

`npm install gatsby-link`

`gatsby-link` auto-detects whether to use a plain `<Link>` or `<NavLink>` based on what props you pass it. There's no need to wrap `<IndexLink>` because it was dropped in react-router v4 in favor of the `exact` prop.

```
import Link from 'gatsby-link'

// Equivalent to react-router's <Link>
<Link to="/page-2/">Page 2</Link>

// Equivalent to react-router's <NavLink>
<Link to="/page-2/" activeClassName="selected">Page 2</Link>

// `exact` prop replaces <IndexLink> from react-router v3
<Link to="/" exact>Home</Link>
```

Prefixing links is also now handled automatically by our new `<Link>` component so remove usages of `prefixLink` in links.

Use `gatsby-link` everywhere and things will Just Work™.

config.toml is now gatsby-config.js

If you previously added site metadata to `config.toml`, move that into the new `gatsby-config.js`.

You need to remove all requires/imports of `config` in your code.

Site-wide metadata should now be "queried" using GraphQL.

A minimal config module would look like:

```
module.exports = {
  siteMetadata: {
    title: `My Sweet Gatsby Site!`,
  },
}
```

and a minimal query would look like

```
export const pageQuery = graphql`
  query SiteMetadataLookup($slug: String!) {
    site {
      siteMetadata {
        title
      }
    }
  }
`
```

exporting that from the same file as a React component will make the config information available to the component as a `data` prop on the component. For instance, the title attribute could be referenced as `props.data.site.siteMetadata.title`.

Migrate wrapper components to template components

In v0, there was the concept of "wrapper" components that would render each file of a given file type. E.g. markdown files would be rendered by a wrapper component at `wrappers/md.js` and JSON files `wrappers/json.js`, etc. Data would be parsed from the files and automatically injected into the wrapper components as props.

If you're *not* using wrappers in your site, feel free to skip this section.

While this proved often intuitive and workable, it was overly prescriptive and restricted the types of pages that could be created due to the required 1-to-1 relationship between files and pages.

So for v1, we're moving to a mode where sites must explicitly create pages and create mappings between template components and data.

Gatsby's new data system turns your data into a queryable database. You can query data in any way to create pages and to pull in data into these pages.

These mappings can range between straightforward and complex. E.g. a markdown blog would want to create a post page for every markdown file. But it also might want to create tag pages for each tag linking to the posts using that tag. See this issue on programmatic routes and this blog post introducing work on v1 for more background on this change.

Here's an example of migrating a markdown wrapper to Gatsby v1.

Add markdown plugins

Install Gatsby plugins for handling markdown files.

`npm install gatsby-source-filesystem@next gatsby-transformer-remark@next gatsby-remark-copy-linked-files@next gatsby-remark-prismjs@next gatsby-remark-responsive-iframe@next`

```
gatsby-remark-images@next gatsby-remark-smartypants@next gatsby-plugin-sharp@next
```

Next add them to your `gatsby-config.js` file. Make your config file look something like the following:

```
module.exports = {
  siteMetadata: {
    title: `My Sweet Gatsby Site!`,
  },
  plugins: [
    {
      resolve: `gatsby-source-filesystem`,
      options: {
        name: `pages`,
        path: `${__dirname}/src/pages/`,
      },
    },
    {
      resolve: `gatsby-transformer-remark`,
      options: {
        plugins: [
          {
            resolve: `gatsby-remark-images`,
            options: {
              maxWidth: 690,
            },
          },
          {
            resolve: `gatsby-remark-responsive-iframe`,
          },
          `gatsby-remark-prismjs`,
          `gatsby-remark-copy-linked-files`,
          `gatsby-remark-smartypants`,
        ],
      },
    },
    `gatsby-plugin-sharp`,
  ],
}
```

Create slugs for markdown files

It's handy to store the pathname or "slug" for each markdown page with the markdown data. This lets you query the slug from multiple places.

Here's how you do that.

```
// In your gatsby-node.js
const path = require("path")

exports.onCreateNode = ({ node, boundActionCreators, getNode }) => {
  const { createNodeField } = boundActionCreators
```

```
  let slug
  if (node.internal.type === `MarkdownRemark`) {
    const fileNode = getNode(node.parent)
    const parsedFilePath = path.parse(fileNode.relativePath)
    if (parsedFilePath.name !== `index` && parsedFilePath.dir !== ``) {
      slug = `/${parsedFilePath.dir}/${parsedFilePath.name}/`
    } else if (parsedFilePath.dir === ``) {
      slug = `/${parsedFilePath.name}/`
    } else {
      slug = `/${parsedFilePath.dir}/`
    }

    // Add slug as a field on the node.
    createNodeField({ node, name: `slug`, value: slug })
  }
}
```

Now you can create pages for each markdown file using our slug. In the same `gatsby-node.js` file add:

```
exports.createPages = ({ graphql, boundActionCreators }) => {
  const { createPage } = boundActionCreators

  return new Promise((resolve, reject) => {
    const pages = []
    const blogPost = path.resolve("src/templates/blog-post.js")
    // Query for all markdown "nodes" and for the slug you previously created.
    resolve(
      graphql(
        `
          {
            allMarkdownRemark {
              edges {
                node {
                  fields {
                    slug
                  }
                }
              }
            }
          }
        `
      ).then(result => {
        if (result.errors) {
          console.log(result.errors)
          reject(result.errors)
        }

        // Create blog posts pages.
        result.data.allMarkdownRemark.edges.forEach(edge => {
          createPage({
            path: edge.node.fields.slug, // required
```

```
            component: blogPost,
            context: {
              slug: edge.node.fields.slug,
            },
          })
        })

        return
      })
    )
  })
}
```

You've now generated the pathname or slug for each markdown page as well as told Gatsby about these pages. You'll notice above that you reference a blog post template file when creating the pages. You haven't created that template file yet, so in your `src` directory, create a templates directory and add `blog-post.js`.

This is a normal React.js component with a special Gatsby twist—a GraphQL query specifying the data needs of the component. As a start, make the component look like the following. You can make it more complex once the basics are working.

```
import React from "react"

class BlogPostTemplate extends React.Component {
  render() {
    const post = this.props.data.markdownRemark

    return (
      <div>
        <h1>{post.frontmatter.title}</h1>
        <div dangerouslySetInnerHTML={{ __html: post.html }} />
      </div>
    )
  }
}

export default BlogPostTemplate

export const pageQuery = graphql`
  query BlogPostBySlug($slug: String!) {
    markdownRemark(fields: { slug: { eq: $slug } }) {
      html
      frontmatter {
        title
      }
    }
  }
`
```

At the bottom of the file you'll notice the GraphQL query. This is how pages and templates in Gatsby v1 get their data. In v0, wrapper components had little control over what data they got. In v1, templates and pages can query for

exactly the data they need.

There will be a more in-depth tutorial and GraphQL-specific documentation soon but in the meantime, check out https://graphql.org/ and play around on Gatsby's built-in GraphQL IDE (Graph_i_QL) which can be reached when you start the development server.

At this point you should have working markdown pages when you run `npm run develop`! Now start gradually adding back what you had in your wrapper component adding HTML elements, styles, and extending the GraphQL query as needed.

Repeat this process for other wrapper components you were using.

html.js

This should generally work the same as in v0 except there are two additional props that must be added to your HTML component. Somewhere in your `<head>` add `{this.props.headComponents}` and somewhere at the end of your body, remove loading `bundle.js` and add `{this.props.postBodyComponents}`.

Also the target div now must have an id of `___gatsby`. So the body section of your `html.js` should look like:

```
<body>
  <div id="___gatsby" dangerouslySetInnerHTML={{ __html: this.props.body }} />
  {this.props.postBodyComponents}
</body>
```

_template.js is now src/layouts/index.js

You should be able to copy your `_template.js` file directly making only one change making `this.props.children` a function call so `this.props.children()`. The rationale for this change is described in this PR comment.

Nested layouts (similar to the nested *template feature) are _not* supported yet but are on the roadmap for v1.