Page HTML Generation

This documentation isn't up to date with the latest version of Gatsby.

Outdated areas are:

- replace mentions of `data.json` with `page-data.json`

You can help by making a PR to update this documentation.

In the previous section, we saw how Gatsby uses webpack to build the JavaScript bundles required to take over the user experience once the first HTML page has finished loading. But how do the original HTML pages get generated?

The high level process is:

1. Create a webpack configuration for Node.js Server Side Rendering (SSR)
2. Build a `render-page.js` that takes a page path and renders its HTML
3. For each page in Redux, call `render-page.js`

webpack

For the first step, we use webpack to build an optimized Node.js bundle. The entry point for this is called `static-entry.js`

static-entry.js

static-entry.js exports a function that takes a path and returns rendered HTML. Here's what it does to create that HTML:

1. Require page, json, and webpack chunk data sources
2. Create HTML React Container
3. Load Page and Data
4. Create Page Component
5. Add Preload Link and Script Tags
6. Inject Page Info to CDATA
7. Render Final HTML Document

1. Require page, json, and webpack chunk data sources

In order to perform the rest of the operations, we need some data sources to work off. These are:

sync-requires.js Exports `components` which is a map of componentChunkName to require statements for the disk location of the component. See Write Out Pages.

data.json Contains all the pages (with componentChunkName, jsonName, and path) and the dataPaths which map jsonName to dataPath. See Write Out Pages for more.

webpack.stats.json Contains a mapping from componentChunkName to the webpack chunks comprising it. See Code Splitting for more.

chunk-map.json Contains a mapping from componentChunkName to their core (non-shared) chunks. See Code Splitting for more.

2. Create HTML React Container

We create an `html` React component that will eventually be rendered to a file. It will have props for each section (e.g. `head`, `preBodyComponents`, `postBodyComponents`). This is owned by default-html.js.

3. Load Page and Data

The only input to `static-entry.js` is a path. So we must look up the page for that path in order to find its `componentChunkName` and `jsonName`. This is achieved by looking up the pages array contained in `data.json`. We can then load its data by looking it up in `dataPaths`.

4. Create Page Component

Now we're ready to create a React component for the page (inside the HTML container). This is handled by RouteHandler. Its render will create an element from the component in `sync-requires.js`.

5. Add Preload Link and Script Tags

This is covered by the Code Splitting docs. We essentially create a `<link rel="preload" href="component.js">` in the document head, and a follow-up `<script src="component.js">` at the end of the document. For each component and page JSON.

6. Inject Page Info to CDATA

The production-app.js needs to know the page that it's rendering. The way we pass this information is by setting it in CDATA during HTML generation, since we know that page at this point. So we add the following to the top of the HTML document:

```
/*
<![
  CDATA[ */
    window.page={
      "path": "/blog/2.js",
      "componentChunkName": "component---src-blog-2-js",
      jsonName": "blog-2-995"
    };
    window.dataPath="621/path---blog-2-995-a74-dwfQIanOJGe2gi27a9CLKHjamc";
  */ ]
]>
*/
```

7. Render Final HTML Document

Finally, we call react-dom and render our top level HTML component to a string and return it.

build-html.js

So, we've built the means to generate HTML for a page. These webpack bundles are saved to `.cache/page-ssr/routes`. Next, we need to use it to generate HTML for all the site's pages.

Page HTML does not depend on other pages. So we can perform this step in parallel. We use the jest-worker library to make this easier. By default, the render-html.ts creates a pool of workers equal to the number of physical cores on your machine. You can configure the number of pools by passing an optional environment variable, `GATSBY_CPU_COUNT`. It then partitions the pages into groups and sends them to the workers, which run worker.

The workers iterate over each page in their partition, and call the `render-page.js` with the page. It then saves the HTML for the page's path in `/public`.

Once all workers have finished, we're done!