Building the JavaScript App

> This documentation isn't up to date with the latest version of Gatsby.
>
> Outdated areas are:
>
> - *Load page resources* section needs to be updated
>
> You can help by making a PR to update this documentation.

Gatsby generates your site's HTML pages, but also creates a JavaScript runtime that takes over in the browser once the initial HTML has loaded. This enables other pages to load instantaneously. Read on to find out how that runtime is generated.

webpack config

The build-javascript.ts Gatsby file is the entry point to this section. It dynamically creates a webpack configuration by calling webpack.config.js. This can produce radically different configs depending on the stage. E.g. `build-javascript`, `build-html`, `develop`, or `develop-html`. This section deals with the `build-javascript` stage.

The config is quite large, but here are some of the important values in the final output.

```
{
  entry: {
    app: `.cache/production-app`
  },
  output: {
    // e.g. app-2e49587d85e03a033f58.js
    filename: `[name]-[contenthash].js`,
    // e.g. component---src-blog-2-js-cebc3ae7596cbb5b0951.js
    chunkFilename: `[name]-[contenthash].js`,
    path: `/public`,
    publicPath: `/`
  },
  target: `web`,
  devtool: `source-map`,
  mode: `production`,
```

```
  node: {
    __filename: true
  },
  optimization: {
    runtimeChunk: {
      // e.g. webpack-runtime-e402cdceeae5fad2aa61.js
      name: `webpack-runtime`
    },
    splitChunks: {
      chunks: `all`,
      cacheGroups: {
        // disable webpack's default cacheGroup
        default: false,
        // disable webpack's default vendor cacheGroup
        vendors: false,
        // Create a framework bundle that contains React libraries
        // They hardly change so we bundle them together to improve
        framework: {},
        // Big modules that are over 160kb are moved to their own file to
        // optimize browser parsing & execution
        lib: {},
        // All libraries that are used on all pages are moved into a common chunk
        commons: {},
        // When a module is used more than once we create a shared bundle to save user's ba
        shared: {},
        // All CSS is bundled into one stylesheet
        styles: {}
      },
      // Keep maximum initial requests to 25
      maxInitialRequests: 25,
      // A chunk should be at least 20kb before using splitChunks
      minSize: 20000
    },
    minimizers: [
      // Minify javascript using Terser (https://terser.org/)
      plugins.minifyJs(),
      // Minify CSS by using cssnano (https://cssnano.co/)
      plugins.minifyCss(),
    ]
  }
  plugins: [
    // A custom webpack plugin that implements logic to write out chunk-map.json and webpac
    plugins.extractStats(),
  ]
}
```

There's a lot going on here. And this is just a sample of the output that doesn't include the loaders, rules, etc. We won't go over everything here, but most of it is geared towards proper code splitting of your application.

The splitChunks section is the most complex part of the Gatsby webpack config as it configures how Gatsby generates the most optimized bundles for your website. This is referred to as Granular Chunks as Gatsby tries to make the generated JavaScript files as granular as possible by deduplicating all modules. You can read more about SplitChunks and chunks on the official webpack website.

Once webpack has finished compilation, it will have produced a few key types of bundles:

app-[contenthash].js

This bundle is produced from production-app.js which will mostly be discussed in this section. It is configured in webpack entry

webpack-runtime-[contenthash].js

This contains the small webpack-runtime as a separate bundle (configured in `optimization` section). In practice, the app and webpack-runtime are always needed together.

framework-[contenthash].js

The framework bundle contains the React framework. Based on user behavior, React hardly gets upgraded to a newer version. Creating a separate bundle improves users' browser cache hit rate as this bundle is likely not going to be updated often.

commons-[contenthash].js

Libraries used on every Gatsby page are bundled into the commons JavaScript file. By bundling these together, you can make sure your users only need to download this bundle once.

component—[name]-[contenthash].js

This is a separate bundle for each page. The mechanics for how these are split off from the main production app are covered in Code Splitting.

production-app.js

This is the entry point to webpack that outputs `app-[contenthash].js` bundle. It is responsible for navigation and page loading once the initial HTML has been loaded.

First load

To show how `production-app` works, let's imagine that you've just refreshed the browser on your site's `/blog/2` page. The HTML loads immediately, painting your page quickly. It includes a CDATA section which injects page information into the `window` object so it's available in your JavaScript code (inserted during Page HTML Generation).

```
/*
<![
  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";
  */ ]
]>
*/
```

Then, the app, webpack-runtime, component, shared libraries, and data JSON bundles are loaded via `<link>` and `<script>` (see HTML tag generation). Now, your `production-app` code starts running.

onClientEntry (api-runner-browser)

The first thing your app does is run the onClientEntry browser API. This allows plugins to perform any operations before you hit the rest of the page loading logic. For example gatsby-plugin-glamor will call rehydrate.

It's worth noting that the browser API runner is completely different to `api-runner-node` which is explained in How APIs/Plugins Are Run. `api-runner-node` runs in Node.js and has to deal with complex server based execution paths. Whereas running APIs on the browser is a matter of iterating through the site's registered browser plugins and running them one after the other (see api-runner-browser.js).

One thing to note is that it gets the list of plugins from `./cache/api-runner-browser-plugins.js`, which is generated early in bootstrap.

DOM hydration

hydrate() is a ReactDOM function which is the same as `render()`, except that instead of generating a new DOM tree and inserting it into the document, it expects that a React DOM already exists with exactly the same structure as the React Model. It therefore descends this tree and attaches the appropriate event listeners to it so that it becomes a live React DOM. Since your HTML was rendered with exactly the same code as you're running in your browser,

these will (and have to) match perfectly. The hydration occurs on the `<div id="___gatsby">...</div>` element defined in default-html.js.

Page rendering

The hydration requires a new React component to "replace" the existing DOM with. Gatsby uses reach router for this. Within it, Gatsby provides a RouteHandler component that uses PageRenderer to create the navigated to page.

PageRenderer's constructor loads the page resources for the path. On first load though, these will have already been requested from the server by `<link rel="preload" ... />` in the page's original HTML (see Link Preloads in HTML Generation Docs). The loaded page resources includes the imported component, with which Gatsby creates the actual page component using React.createElement(). This element is returned to the RouteHandler which hands it off to Reach Router for rendering.

Load page resources

Before hydration occurs, Gatsby kicks off the loading of resources in the background. As mentioned above, the current page's resources will have already been requested by `link` tags in the HTML. So, technically, there's nothing more required for this page load. But we can start loading resources required to navigate to other pages.

This occurs in loader.js. The main function here is getResourcesForPathname(). Given a path, it will find its page, and import its component module JSON query results. But to do this, it needs access to that information. This is provided by async-requires.js which contains the list of all pages in the site, and all their dataPaths. fetchPageResourcesMap() takes care of requesting that file, which occurs the first time `getResourcesForPathname()` is called.

`window` variables

Gatsby attaches global state to the `window` object via `window.___somevar` variables so they can be used by plugins (though this is technically unsupported). Here are a few:

`___loader` This is a reference to the loader.js object that can be used for getting page resources and enqueueing prefetch commands. It is used by gatsby-link to prefetch pages. And by gatsby-plugin-guess-js to implement its own prefetching algorithm.

`___emitter` Only used during `gatsby develop` lifecycle

`___chunkMapping` Contents of `chunk-map.json`. See Code Splitting for more.

`___push`, `___replace` and `___navigate` These are set in init navigation. Used by `gatsby-link` to override navigation behavior so that it loads pages before using reach to navigate.