This is a high-level overview about the steps in the Gatsby build process. For more detailed information about specific steps, check out the Gatsby Internals section of the docs.

Gatsby has two modes for compiling a site:

1. Develop - run with the `gatsby develop` command
2. Build - run with `gatsby build`

You can start Gatsby in either mode with its respective command: `gatsby develop` or `gatsby build`.

Build time vs runtime

A common confusion with the generation of static assets is the difference between build time and runtime.

Processes that happen in a web browser that you click through and interact with can be referred to as a *browser runtime*. JavaScript code can interact with the browser and take advantage of APIs it offers, such as `window.location` to get information from a page's URL, using AJAX to submit a form, or injecting markup dynamically into the DOM. Gatsby creates a JavaScript runtime that takes over in the browser once the initial HTML has loaded.

Build time, in contrast, refers to the process of using a server process to compile the site into files that can be delivered to a web browser later, so Browser APIs like `window` aren't available at that time.

The `gatsby develop` command doesn't perform some of the production build steps that the `gatsby build` command does. Instead, it starts up a development server that you can use to preview your site in the browser -- like a runtime. When you run `gatsby build` (build time) there won't be a browser available, so your site needs to be capable of protecting calls to browser based APIs.

To gain a greater understanding of what happens when you run either command, it can be helpful to look at the information that Gatsby is reporting back and break it down.

Understanding `gatsby develop` (runtime)

Using `gatsby develop` runs a server in the background, enabling useful features like live reloading and Gatsby's data explorer.

`gatsby develop` is optimized for rapid feedback and extra debugging information. The output of running `gatsby develop` in a fresh install of the Gatsby default starter looks like this:

```
success open and validate gatsby-configs - 0.051 s
success load plugins - 0.591 s
success onPreInit - 0.015 s
success initialize cache - 0.019 s
success copy gatsby files - 0.076 s
success onPreBootstrap - 0.021 s
success source and transform nodes - 0.082 s
success Add explicit types - 0.018 s
success Add inferred types - 0.106 s
success Processing types - 0.080 s
success building schema - 0.266 s
success createPages - 0.014 s
success createPagesStatefully - 0.067 s
success onPreExtractQueries - 0.017 s
success update schema - 0.034 s
success extract queries from components - 0.222 s
success write out requires - 0.044 s
success write out redirect data - 0.014 s
success Build manifest and related icons - 0.110 s
success onPostBootstrap - 0.130 s

info bootstrap finished - 3.674 s

success run static queries - 0.057 s — 3/3 89.08 queries/second
success run page queries - 0.026s - 3/3 114.85/s
success start webpack server - 1.707 s — 1/1 6.06 pages/second
```

Some of the steps may be self-explanatory, but others require context of Gatsby internals to make complete sense of (don't worry, the rest of this guide will give more explanations).

Understanding `gatsby build` (build time)

Gatsby's `build` command should be run when you've added the finishing touches to your site and everything looks great. `gatsby build` creates a version of your site with production-ready optimizations like packaging up your site's config, data, and code, and creating all the static HTML pages that eventually get rehydrated into a React application.

The output of running `gatsby build` in a fresh install of the Gatsby default starter looks like this:

```
success open and validate gatsby-configs - 0.062 s
success load plugins - 0.915 s
success onPreInit - 0.021 s
success delete html and css files from previous builds - 0.030 s
success initialize cache - 0.034 s
success copy gatsby files - 0.099 s
success onPreBootstrap - 0.034 s
success source and transform nodes - 0.121 s
success Add explicit types - 0.025 s
success Add inferred types - 0.144 s
success Processing types - 0.110 s
success building schema - 0.365 s
success createPages - 0.016 s
success createPagesStatefully - 0.079 s
success onPreExtractQueries - 0.025 s
success update schema - 0.041 s
success extract queries from components - 0.333 s
success write out requires - 0.020 s
success write out redirect data - 0.019 s
success Build manifest and related icons - 0.141 s
success onPostBootstrap - 0.164 s

info bootstrap finished - 6.932 s

success run static queries - 0.166 s — 3/3 20.90 queries/second
success Generating image thumbnails — 6/6 - 1.059 s
success Building production JavaScript and CSS bundles - 8.050 s
success Rewriting compilation hashes - 0.021 s
success run page queries - 0.034 s — 4/4 441.23 queries/second
success Building static HTML for pages - 0.852 s — 4/4 23.89 pages/second
info Done building in 16.143999152 sec
```

Differences between develop and build

So what's the difference?

If you compare the outputs of the two commands (`gatsby develop` vs `gatsby build`), you can see that everything (with the exception of deleting HTML and CSS files) up until the line that says `info bootstrap finished` are the same. However, `gatsby build` runs some additional steps to prepare your site to go live after the bootstrap phase.

The following output shows the differences between the above two examples:

```diff
success open and validate gatsby-configs - 0.051 s
success load plugins - 0.915 s
success onPreInit - 0.021 s
+ success delete html and css files from previous builds - 0.030 s
success initialize cache - 0.034 s
success copy gatsby files - 0.099 s
success onPreBootstrap - 0.034 s
success source and transform nodes - 0.121 s
success Add explicit types - 0.025 s
success Add inferred types - 0.144 s
success Processing types - 0.110 s
success building schema - 0.365 s
success createPages - 0.016 s
success createPagesStatefully - 0.079 s
success onPreExtractQueries - 0.025 s
success update schema - 0.041 s
success extract queries from components - 0.333 s
success write out requires - 0.020 s
success write out redirect data - 0.019 s
success Build manifest and related icons - 0.141 s
success onPostBootstrap - 0.130 s

info bootstrap finished - 3.674 s

success run static queries - 0.057 s — 3/3 89.08 queries/second
- success run page queries - 0.033 s — 5/5 347.81 queries/second
- success start webpack server - 1.707 s — 1/1 6.06 pages/second
+ success run page queries - 0.026s - 3/3 114.85/s
+ success Generating image thumbnails — 6/6 - 1.059 s
+ success Building production JavaScript and CSS bundles - 8.050 s
+ success Rewriting compilation hashes - 0.021 s
+ success Building static HTML for pages - 0.852 s — 4/4 23.89 pages/second
+ info Done building in 16.143999152 sec
```

Note: *the output of* `gatsby develop` *and* `gatsby build` *can vary based on plugins you've installed that can tap into the build lifecycle and the Gatsby reporter. The above output is from running the default Gatsby starter.*

There is only one difference in the bootstrap phase where HTML and CSS is deleted to prevent problems with previous builds. In the build phase, the build command skips setting up a dev server and goes into compiling the assets.

By omitting these later steps, `gatsby develop` can speed up your ability to make live edits without page reloads using features like hot module replacement. It also saves time with the more CPU intensive processes that aren't necessary to perform in rapid development.

There is also a difference in the number of page queries that will run. `gatsby develop` will run at most 3 page queries (index page, actual 404 and develop 404) initially. The rest of the queries will run when they are needed (when the browser requests them). In contrast, `gatsby build` will run page queries for every page that doesn't have cached and up to date results already.

A cache is also used to detect changes to `gatsby-*.js` files (like `gatsby-node.js`, or `gatsby-config.js`) or dependencies. It can be cleared manually with the `gatsby clean` command to work through issues caused by outdated references and a stale cache.

What happens when you run `gatsby build`?

To see the code where many of these processes are happening, refer to the code and comments in the `build` and `bootstrap` files of the repository.

A Node.js server process powers things behind the scenes when you run the `gatsby build` command. The process of converting assets and pages into HTML that can be rendered in a browser by a server-side language like Node.js is referred to as server-side rendering, or SSR. Since Gatsby builds everything ahead of time, this creates an entire site with all of the data your pages need at once. When the site is deployed, it doesn't need to run with server-side processes because everything has been gathered up and compiled by Gatsby.

Note: because Gatsby sites run as full React applications in the browser, you can still fetch data from other sources at runtime like you would in a typical React app.

The following model demonstrates what is happening in different "layers" of Gatsby as content and data are gathered up and made available for your static assets.

Like the console output demonstrates in the section above, there are 2 main steps that take place when you run a build: the `bootstrap` phase, and the `build` phase (which can be observed finishing in the console output when you run `gatsby develop` or `gatsby build`).

```
info bootstrap finished - 3.674 s
...
info Done building in 16.143999152 sec
```

At a high level, what happens during the whole bootstrap and build process is:

1. `Node` objects are sourced from whatever sources you defined in `gatsby-config.js` with plugins as well as in your `gatsby-node.js` file
2. A schema is inferred from the Node objects
3. Pages are created based off JavaScript components in your site or in installed themes
4. GraphQL queries are extracted and run to provide data for all pages

5. Static files are created and bundled in the `public` directory

For an introduction to what happens at each step throughout the process from the output above, refer to the sections below.

Steps of the bootstrap phase

The steps of the bootstrap phase are shared between develop and build (with only one exception in the `delete html and css files from previous builds` step). This includes:

1. `open and validate gatsby-configs`

The `gatsby-config.js` file for the site and any installed themes are opened, ensuring that a function or object is exported for each.

2. `load plugins`

Plugins installed and included in the config of your site and your site's themes are loaded. Gatsby uses Redux for state management internally and stores info like the version, name, and what APIs are used by each plugin.

3. `onPreInit`

Runs the `onPreInit` node API if it has been implemented by your site or any installed plugins.

4. `delete html and css files from previous builds`

The only different step between develop and build, the HTML and CSS from previous builds is deleted to prevent problems with styles and pages that no longer exist.

5. `initialize cache`

Check if new dependencies have been installed in the `package.json`; if the versions of installed plugins have changed; or if the `gatsby-config.js` or the `gatsby-node.js` files have changed. Plugins can interact with the cache.

6. `copy gatsby files`

Copies site files into the cache in the `.cache` folder.

7. `onPreBootstrap`

Calls the `onPreBootstrap` node API in your site or plugins where it is implemented.

8. `source and transform nodes`

Creates Node objects from your site and all plugins implementing the `sourceNodes` API, and warns about plugins that aren't creating any nodes. Nodes created by source or transformer plugins are cached.

Node objects created at this stage are considered top level nodes, meaning they don't have a parent node that they are derived from.

9. `Add explicit types`

Adds types to the GraphQL schema for nodes that you have defined explicitly with Gatsby's schema optimization APIs.

10. `Add inferred types`

All other nodes not already defined are inspected and have types inferred by Gatsby.

11. `Processing types`

Composes 3rd party schema types, child fields, custom resolve functions, and sets fields in the GraphQL schema. It then prints information about type definitions.

12. `building schema`

Imports the composed GraphQL schema and builds it.

13. `createPages`

Calls the `createPages` API for your site and all plugins implementing it, like when you create pages programmatically in your `gatsby-node.js` .

Plugins can handle the `onCreatePage` event at this point for use cases like manipulating the path of pages.

14. `createPagesStatefully`

Similar to the `createPages` step, but for the `createPagesStatefully` API which is intended for plugins who want to manage creating and removing pages in response to changes in data not managed by Gatsby.

15. `onPreExtractQueries`

Calls the `onPreExtractQueries` API for your site and all plugins implementing it.

16. `update schema`

Rebuilds the GraphQL schema, this time with `SitePage` context -- an internal piece of Gatsby that allows you to introspect all pages created for your site.

17. `extract queries from components`

All JavaScript files in the site are loaded and Gatsby determines if there are any GraphQL queries exported from them. If there are problematic queries they can be reported back with warnings or errors. All these queries get queued up for execution in a later step.

18. `write out requires`

An internal Gatsby utility adds the code that files need to load/require.

19. `write out redirect data`

An internal Gatsby utility adds code for redirects, like implemented with `createRedirect` .

20. `Build manifest and related icons` - (from `gatsby-plugin-manifest`)

This step is activated by `gatsby-plugin-manifest` in the `gatsby-default-starter` and is not a part of the built-in Gatsby functionality, demonstrating how plugins are able to tap into the lifecycle. The plugin adds a `manifest.json` file with the specified configurations and icons.

21. `onPostBootstrap`

Calls the `onPostBootstrap` API for your site and all plugins implementing it.

Steps of the build phase

1. `run static queries`

Static queries in non-page components that were queued up earlier from query extraction are run to provide data to the components that need it.

2. `Generating image thumbnails — 6/6` - (from `gatsby-plugin-sharp`)

Another step that is not a part of the built-in Gatsby functionality, but is the result of installing `gatsby-plugin-sharp`, which taps into the lifecycle. Sharp runs processing on images to create image assets of different sizes.

3. `Building production JavaScript and CSS bundles`

Compiles JavaScript and CSS using webpack.

4. `Rewriting compilation hashes`

Compilation hashes are used by webpack for code splitting and keeping the cache up to date, and all files with page data need to be updated with the new hashes since they've been recompiled.

5. `run page queries`

Page queries that were queued up earlier from query extraction are run so the data pages need can be provided to them.

6. `Building static HTML for pages`

With everything ready for the HTML pages in place, HTML is compiled and written out to files so it can be served up statically. Since HTML is being produced in a Node.js server context, references to browser APIs like `window` can break the build and must be conditionally applied.

Gatsby will smartly rebuild only needed HTML files. This can result in no HTML files being generated if nothing used for HTML files changed. The opposite can also be true and lead to a full site rebuild when data is used on all pages or when a code change happens.

What do you get from a successful build?

When a Gatsby build is successfully completed, everything you need to deploy your site ends up in the `public` folder at the root of the site. The build includes minified files, transformed images, JSON files with information and data for each page, static HTML for each page, and more.

The final build is just static files so it can now be deployed.

You can run `gatsby serve` to test the output from compilation. If you make any new changes to the source code that you want to preview, you will have to rebuild the site and run `gatsby serve` again.