Gatsby's APIs are tailored conceptually to some extent after React.js to improve the coherence between the two systems.

The two top priorities of the API are : a) enable a broad and robust plugin ecosystem and b) build on top of that a broad and robust theme ecosystem.

Prerequisites

If you're not familiar with Gatsby's lifecycle, see the overview Gatsby Lifecycle APIs.

Plugins

Plugins can extend Gatsby in many ways:

- Sourcing data (e.g. from the filesystem or an API or a database)
- Transforming data from one type to another (e.g. a markdown file to HTML)
- Creating pages (e.g. a directory of markdown files all gets turned into pages with URLs derived from their file names).
- Modifying webpack config (e.g. for styling options, adding support for other compile-to-js languages)
- Adding things to the rendered HTML (e.g. meta tags, analytics JS snippets like Google Analytics)
- Writing out things to build directory based on site data (e.g. service worker, sitemap, RSS feed)

A single plugin can use multiple APIs to accomplish its purpose. E.g. the plugin for the CSS-in-JS library Glamor:

1. modifies the webpack config to add its plugin
2. adds a Babel plugin to replace React's default createElement
3. modifies server rendering to extract out the critical CSS for each rendered page and inline the CSS in the `<head>` of that HTML page.

Plugins can also depend on other plugins. The Sharp plugin exposes a number of high-level APIs for transforming images that several other Gatsby image plugins depend on. gatsby-transformer-remark does basic markdown->HTML transformation but exposes an API to allow other plugins to intervene in the conversion process e.g. gatsby-remark-prismjs which adds highlighting to code blocks.

Transformer plugins are decoupled from source plugins. Transformer plugins look at the media type of new nodes created by source plugins to decide if they can transform it or not. Which means that a markdown transformer plugin can transform markdown from any source without any other configuration e.g. from a file, a code comment, or external service like Trello which supports markdown in some of its data fields.

See the full list of (official only for now — adding support for community plugins later) plugins.

API

Concepts

- *Page* — a site page with a pathname, a template component, and optional GraphQL query.
- *Page Component* — React.js component that renders a page and can optionally specify a GraphQL query
- *Component extensions* — extensions that are resolvable as components. `.js` and `.jsx` are supported by core. But plugins can add support for other compile-to-js languages.
- *Dependency* — Gatsby automatically tracks dependencies between different objects e.g. a page can depend on certain nodes. This allows for hot reloading, caching, incremental rebuilds, etc.
- *Node* — a data object
- *Node Field* — a field added by a plugin to a node that it doesn't control

- *Node Link* — a connection between nodes that gets converted to GraphQL relationships. Can be created in a variety of ways as well as automatically inferred. Parent/child links from nodes and their transformed derivative nodes are first class links.

More definitions and terms are defined in the Glossary

Operators

- *Create* — make a new thing
- *Get* — get an existing thing
- *Delete* — remove an existing thing
- *Replace* — replace an existing thing
- *Set* — merge into an existing thing

Extension APIs

Gatsby has multiple processes. The most prominent is the "bootstrap" process. It has several subprocesses. One tricky part to their design is that they run both once during the initial bootstrap but also stay alive during development to continue to respond to changes. This is what drives hot reloading that all Gatsby data is "alive" and reacts to changes in the environment.

The bootstrap process is as follows:

load site config -> load plugins -> source nodes -> transform nodes -> create GraphQL schema -> create pages -> compile component queries -> run queries -> fin

Once the initial bootstrap is finished, a `webpack-dev-server` and express server are started for serving files for the development workflow with live updates. For a production build, Gatsby skips the development server and instead builds the CSS, then JavaScript, then static HTML with webpack.

During these processes there are various extension points where plugins can intervene. All major processes have an `onPre` and `onPost` e.g. `onPreBootstrap` and `onPostBootstrap` or `onPreBuild` or `onPostBuild`. During bootstrap plugins can respond at various stages to APIs like `onCreatePages`, `onCreateBabelConfig`, and `onSourceNodes`.

At each extension point, Gatsby identifies the plugins which implement the API and calls them in serial following their order in the site's `gatsby-config.js`.

In addition to extension APIs in a node, plugins can also implement extension APIs in the server rendering process and the browser e.g. `onClientEntry` or `onRouteUpdate`.

The three main inspirations for this API and spec are React.js' API specifically @leebyron's email on the React API, this talk "How to Design a Good API and Why it Matters" by Joshua Bloch who designed many parts of Java, and Hapi.js' plugin design.