During Gatsby's bootstrap & build phases, the state is stored and manipulated using the Redux library. The key purpose of using Redux in Gatsby internals is to centralize all of the state logic. Reviewing the Gatsby reducers and actions folders gives a comprehensive picture of what state manipulations are possible.

Store

The namespaces in Gatsby's Redux store are a great overview of the Gatsby internals, here are a few:

- **Nodes** - All data that's added to Gatsby is modeled using nodes. Nodes are most commonly added by Source plugins such as `gatsby-source-filesystem`.
- **Schema** - GraphQL schema inferred from Nodes, available for querying by page and static queries.
- **Pages** - A `Map` of page paths to page objects. Objects made via Page Creation contain information needed to render a page such as component file path, page query and context.
- **Components** - A `Map` of component file paths to page objects.
- **Static Query Components** - A `Map` of any components detected with a static query.
- **Jobs** - Long-running and CPU-intensive processes, generally started as a side effect to a GraphQL query. Gatsby doesn't finish its process until all jobs are ended.
- **webpack** - Config for the webpack bundler which handles code optimization and splitting of delivered JavaScript bundles.

The Gatsby Redux index file has two key exports, `store` and `emitter`. Throughout the bootstrap and build phases, `store` is used to get the current state and dispatch actions, while `emitter` is used to register listeners for particular actions. The store is also made available to Gatsby users through the Node APIs.

Actions

Actions dispatched in the store cause state changes through the reducers and also trigger listeners registered for that action on a mitt `emitter`. While the `subscribe` Redux store method is typically used to connect a web framework like React, Gatsby only uses the `subscribe` method to connect the `emitter`.

The Gatsby actions are all either internal, public or restricted. The public actions, and a context relevant subset of the restricted actions, are available to users through the Node APIs.

Example action journey for `createRedirect`

Gatsby actions have a similar journey through defining, exposing and dispatching. This section follows the createRedirect public action:

- **Reducer case** - The redirects reducer will catch actions with a type `CREATE_REDIRECT` and make the necessary state manipulation.
- **Side effect** - An `emitter` listener is registered for the `CREATE_REDIRECT` action type.
- **Action creator** - An action creator, `createRedirect`, is defined in the public actions file. The action has a payload, the information needed to complete the action, and a type, the string that identifies this particular action.
- **Expose bound action creator** - `createRedirect` is one of the public actions made available to all of the Node APIs. A collection of public actions and the restricted actions available to the called API are bound to the Redux store dispatch. The bound action collection is then passed when calling the user's API function.
- **Dispatch** - Here is an example of the `createRedirect` call that a Gatsby user could make with the createPages API in their project's gatsby-node.js file:

```
module.exports = {
  createPages: ({ actions }) => {
    const { createRedirect } = actions
    createRedirect({
      fromPath: "/legacy-path",
      toPath: "/current-path",
    })
  },
}
```

By walking through an action scenario in detail, you can hopefully understand more about Gatsby's internals using Redux.

Additional resources

- Behind the Scenes: What Makes Gatsby Great
- Gatsby Jargon