Adding Search with JS Search

Prerequisites

Before you go through the steps needed for adding client-side search to your Gatsby website, you should be familiar with the basics of Gatsby. Check out the tutorial and brush up on the documentation if you need to. In addition, some knowledge of ES6 syntax will be useful.

What is JS Search

JS Search is a library created by Brian Vaughn, a member of the core team at Facebook. It provides an efficient way to search for data on the client with JavaScript and JSON objects. It also has extensive customization options, check out their docs for more details.

The full code and documentation for this library is available on GitHub. This guide is based on the official `js-search` example but has been adapted to work with your Gatsby site.

Setup

You'll start by creating a new Gatsby site based on the official *hello world* starter. Open up a terminal and run the following command:

```
gatsby new js-search-example https://github.com/gatsbyjs/gatsby-starter-default
```

After the process is complete, some additional packages are needed.

Change directories to the `js-search-example` folder and issue the following command:

```
npm install js-search axios
```

Or if Yarn is being used:

```
yarn add js-search axios
```

Note:

For this particular example axios will be used to handle all of the promise-based HTTP requests.

Strategy selection

In the next sections you'll learn about two approaches to implementing `js-search` in your site. Which one you choose will depend on the number of items you want to search. For a small to medium dataset, the first strategy documented should work out nicely.

For larger datasets you could use the second approach, as most of the work is done beforehand through the use of Gatsby's internal API.

Both ways are fairly generalistic, they were implemented using the default options for the library, so that it can be experimented without going through into the specifics of the library.

And finally, as you go through the code, be mindful it does not adhere to the best practices, it's just for demonstration purposes, in a real site it would have been implemented in a different way.

JS-Search with a small to medium dataset

Start by creating a file named `SearchContainer.js` in the `src/components/` folder, then add the following code to get started:

```
import React, { Component } from "react"
import Axios from "axios"
import * as JsSearch from "js-search"

class Search extends Component {
  state = {
    bookList: [],
    search: [],
    searchResults: [],
    isLoading: true,
    isError: false,
    searchQuery: "",
  }
  /**
   * React lifecycle method to fetch the data
   */
  async componentDidMount() {
    Axios.get("https://bvaughn.github.io/js-search/books.json")
      .then(result => {
        const bookData = result.data
        this.setState({ bookList: bookData.books })
        this.rebuildIndex()
      })
      .catch(err => {
        this.setState({ isError: true })
        console.log("====================================")
        console.log(`Something bad happened while fetching the data\n${err}`)
        console.log("====================================")
      })
  }

  /**
   * rebuilds the overall index based on the options
   */
  rebuildIndex = () => {
    const { bookList } = this.state
    const dataToSearch = new JsSearch.Search("isbn")
    /**
     * defines an indexing strategy for the data
     * more about it in here https://github.com/bvaughn/js-search#configuring-the-index-stra
     */
    dataToSearch.indexStrategy = new JsSearch.PrefixIndexStrategy()
    /**
     * defines the sanitizer for the search
     * to prevent some of the words from being excluded
     *
     */
    dataToSearch.sanitizer = new JsSearch.LowerCaseSanitizer()
    /**
     * defines the search index
     * read more in here https://github.com/bvaughn/js-search#configuring-the-search-index
     */
    dataToSearch.searchIndex = new JsSearch.TfIdfSearchIndex("isbn")

    dataToSearch.addIndex("title") // sets the index attribute for the data
    dataToSearch.addIndex("author") // sets the index attribute for the data

    dataToSearch.addDocuments(bookList) // adds the data to be searched
    this.setState({ search: dataToSearch, isLoading: false })
  }

  /**
   * handles the input change and perform a search with js-search
   * in which the results will be added to the state
   */
  searchData = e => {
    const { search } = this.state
    const queryResult = search.search(e.target.value)
    this.setState({ searchQuery: e.target.value, searchResults: queryResult })
  }
  handleSubmit = e => {
    e.preventDefault()
  }

  render() {
    const { bookList, searchResults, searchQuery } = this.state
    const queryResults = searchQuery === "" ? bookList : searchResults
    return (
      <div>
        <div style={{ margin: "0 auto" }}>
          <form onSubmit={this.handleSubmit}>
            <div style={{ margin: "0 auto" }}>
              <label htmlFor="Search" style={{ paddingRight: "10px" }}>
                Enter your search here
              </label>
              <input
                id="Search"
                value={searchQuery}
                onChange={this.searchData}
                placeholder="Enter your search here"
                style={{ margin: "0 auto", width: "400px" }}
              />
            </div>
          </form>
          <div>
            Number of items:
            {queryResults.length}
            <table
              style={{
                width: "100%",
                borderCollapse: "collapse",
                borderRadius: "4px",
                border: "1px solid #d3d3d3",
              }}
            >
              <thead style={{ border: "1px solid #808080" }}>
                <tr>
                  <th
                    style={{
                      textAlign: "left",
                      padding: "5px",
                      fontSize: "14px",
                      fontWeight: 600,
                      borderBottom: "2px solid #d3d3d3",
                      cursor: "pointer",
                    }}
                  >
            Book ISBN
          </th>
          <th
            style={{
              textAlign: "left",
              padding: "5px",
              fontSize: "14px",
              fontWeight: 600,
              borderBottom: "2px solid #d3d3d3",
              cursor: "pointer",
            }}
          >
            Book Title
          </th>
          <th
            style={{
              textAlign: "left",
              padding: "5px",
              fontSize: "14px",
              fontWeight: 600,
              borderBottom: "2px solid #d3d3d3",
              cursor: "pointer",
            }}
          >
            Book Author
          </th>
        </tr>
      </thead>
      <tbody>
        {queryResults.map(item => {
          return (
            <tr key={`row_${item.isbn}`}>
              <td
                style={{
                  fontSize: "14px",
                  border: "1px solid #d3d3d3",
                }}
              >
                {item.isbn}
              </td>
              <td
                style={{
                  fontSize: "14px",
                  border: "1px solid #d3d3d3",
                }}
              >
```

```
                  {item.title}
                </td>
                <td
                  style={{
                    fontSize: "14px",
                    border: "1px solid #d3d3d3",
                  }}
                >
                  {item.author}
                </td>
              </tr>
            )
          })}
        </tbody>
      </table>
    </div>
  </div>
</div>
    )
  }
}
export default Search
```

Breaking down the code into smaller parts:

1. When the component is mounted, the `componentDidMount()` lifecycle method is triggered and the data will be fetched.
2. If no errors occur, the data received is added to the state and the `rebuildIndex()` function is invoked.
3. The search engine is then created and configured with the default options.
4. The data is then indexed using js-search.
5. When the contents of the input changes, js-search starts the search process based on the `input`'s value and returns the search results if any, which is then presented to the user via the `table` element.

Joining all the pieces

In order to get it working in your site, you would only need to import the newly created component to a page. As you can see in the example site.

Run `gatsby develop` and if all went well, open your browser of choice and enter the url `http://localhost:8000` - you'll have a fully functional search at your disposal.

JS-Search with a big dataset

Now try a different approach, this time instead of letting the component do all of the work, it's Gatsby's job to do that and pass all the data to a page defined by the path property, via pageContext.

To do this, some changes are required.

Start by modifying the `gatsby-node.js` file by adding the following code:

```
const path = require("path")
const axios = require("axios")

exports.createPages = ({ actions }) => {
  const { createPage } = actions
  return new Promise((resolve, reject) => {
    axios
      .get("https://bvaughn.github.io/js-search/books.json")
      .then(result => {
        const { data } = result
        /**
         * creates a dynamic page with the data received
         * injects the data into the context object alongside with some options
         * to configure js-search
         */
        createPage({
          path: "/search",
          component: path.resolve(`./src/templates/ClientSearchTemplate.js`),
          context: {
            bookData: {
              allBooks: data.books,
              options: {
                indexStrategy: "Prefix match",
                searchSanitizer: "Lower Case",
                TitleIndex: true,
                AuthorIndex: true,
                SearchByTerm: true,
              },
            },
          },
        })
        resolve()
      })
      .catch(err => {
        console.log("====================================")
        console.log(`error creating Page:${err}`)
        console.log("====================================")
        reject(new Error(`error on page creation:\n${err}`))
      })
  })
}
```

Create a file named `ClientSearchTemplate.js` in the `src/templates/` folder, then add the following code to get started:

```
import React from "react"
import ClientSearch from "../components/ClientSearch"

const SearchTemplate = props => {
  const { pageContext } = props
  const { bookData } = pageContext
  const { allBooks, options } = bookData
  return (
    <div>
      <h1 style={{ marginTop: `3em`, textAlign: `center` }}>
        Search data using JS Search using Gatsby API
      </h1>
      <div>
        <ClientSearch books={allBooks} engine={options} />
      </div>
    </div>
  )
}

export default SearchTemplate
```

Create a file named `ClientSearch.js` in the `src/components/` folder, then add the following code as a baseline:

```
import React, { Component } from "react"
import * as JsSearch from "js-search"

class ClientSearch extends Component {
  state = {
    isLoading: true,
    searchResults: [],
    search: null,
    isError: false,
    indexByTitle: false,
    indexByAuthor: false,
    termFrequency: true,
    removeStopWords: false,
    searchQuery: "",
    selectedStrategy: "",
    selectedSanitizer: "",
  }
  /**
   * React lifecycle method that will inject the data into the state.
   */
```

```
static getDerivedStateFromProps(nextProps, prevState) {
  if (prevState.search === null) {
    const { engine } = nextProps
    return {
      indexByTitle: engine.TitleIndex,
      indexByAuthor: engine.AuthorIndex,
      termFrequency: engine.SearchByTerm,
      selectedSanitizer: engine.searchSanitizer,
      selectedStrategy: engine.indexStrategy,
    }
  }
  return null
}
async componentDidMount() {
  this.rebuildIndex()
}

/**
 * rebuilds the overall index based on the options
 */
rebuildIndex = () => {
  const {
    selectedStrategy,
    selectedSanitizer,
    removeStopWords,
    termFrequency,
    indexByTitle,
    indexByAuthor,
  } = this.state
  const { books } = this.props

  const dataToSearch = new JsSearch.Search("isbn")

  if (removeStopWords) {
    dataToSearch.tokenizer = new JsSearch.StopWordsTokenizer(
      dataToSearch.tokenizer
    )
  }
  /**
   * defines an indexing strategy for the data
   * read more about it here https://github.com/bvaughn/js-search#configuring-the-index-s
   */
  if (selectedStrategy === "All") {
    dataToSearch.indexStrategy = new JsSearch.AllSubstringsIndexStrategy()
  }
  if (selectedStrategy === "Exact match") {
    dataToSearch.indexStrategy = new JsSearch.ExactWordIndexStrategy()
  }
  if (selectedStrategy === "Prefix match") {
    dataToSearch.indexStrategy = new JsSearch.PrefixIndexStrategy()
  }

  /**
   * defines the sanitizer for the search
   * to prevent some of the words from being excluded
   */
  selectedSanitizer === "Case Sensitive"
    ? (dataToSearch.sanitizer = new JsSearch.CaseSensitiveSanitizer())
    : (dataToSearch.sanitizer = new JsSearch.LowerCaseSanitizer())
  termFrequency === true
    ? (dataToSearch.searchIndex = new JsSearch.TfIdfSearchIndex("isbn"))
    : (dataToSearch.searchIndex = new JsSearch.UnorderedSearchIndex())

  // sets the index attribute for the data
  if (indexByTitle) {
    dataToSearch.addIndex("title")
  }
  // sets the index attribute for the data
  if (indexByAuthor) {
    dataToSearch.addIndex("author")
  }

  dataToSearch.addDocuments(books) // adds the data to be searched

  this.setState({ search: dataToSearch, isLoading: false })
}
/**
 * handles the input change and perform a search with js-search
 * in which the results will be added to the state
 */
searchData = e => {
  const { search } = this.state
  const queryResult = search.search(e.target.value)
  this.setState({ searchQuery: e.target.value, searchResults: queryResult })
}
handleSubmit = e => {
  e.preventDefault()
}
render() {
  const { searchResults, searchQuery } = this.state
  const { books } = this.props
  const queryResults = searchQuery === "" ? books : searchResults
```

```
return (
  <div>
    <div style={{ margin: "0 auto" }}>
      <form onSubmit={this.handleSubmit}>
        <div style={{ margin: "0 auto" }}>
          <label htmlFor="Search" style={{ paddingRight: "10px" }}>
            Enter your search here
          </label>
          <input
            id="Search"
            value={searchQuery}
            onChange={this.searchData}
            placeholder="Enter your search here"
            style={{ margin: "0 auto", width: "400px" }}
          />
        </div>
      </form>
      <div>
        Number of items:
        {queryResults.length}
        <table
          style={{
            width: "100%",
            borderCollapse: "collapse",
            borderRadius: "4px",
            border: "1px solid #d3d3d3",
          }}
        >
          <thead style={{ border: "1px solid #808080" }}>
            <tr>
              <th
                style={{
                  textAlign: "left",
                  padding: "5px",
                  fontSize: "14px",
                  fontWeight: 600,
                  borderBottom: "2px solid #d3d3d3",
                  cursor: "pointer",
                }}
              >
                Book ISBN
              </th>
              <th
                style={{
                  textAlign: "left",
                  padding: "5px",
        fontSize: "14px",
        fontWeight: 600,
        borderBottom: "2px solid #d3d3d3",
        cursor: "pointer",
      }}
    >
      Book Title
    </th>
    <th
      style={{
        textAlign: "left",
        padding: "5px",
        fontSize: "14px",
        fontWeight: 600,
        borderBottom: "2px solid #d3d3d3",
        cursor: "pointer",
      }}
    >
      Book Author
    </th>
  </tr>
</thead>
<tbody>
  {queryResults.map(item => {
    return (
      <tr key={`row_${item.isbn}`}>
        <td
          style={{
            fontSize: "14px",
            border: "1px solid #d3d3d3",
          }}
        >
          {item.isbn}
        </td>
        <td
          style={{
            fontSize: "14px",
            border: "1px solid #d3d3d3",
          }}
        >
          {item.title}
        </td>
        <td
          style={{
            fontSize: "14px",
            border: "1px solid #d3d3d3",
                        }}
                      >
                        {item.author}
                      </td>
                    </tr>
                  )
                })}
              </tbody>
            </table>
          </div>
        </div>
      </div>
    )
  }
}
export default ClientSearch
```

Breaking down the code into smaller parts:

1. When the component is mounted, the `getDerivedStateFromProps()` lifecycle method is invoked and it will evaluate the state and if necessary update it.
2. Then the `componentDidMount()` lifecycle method will be triggered and the `rebuildIndex()` function is invoked.
3. The search engine is then created and configured with the options defined.
4. The data is then indexed using js-search.
5. When the contents of the input changes, js-search starts the search process based on the `input`'s value and returns the search results if any, which is then presented to the user via the `table` element.

Joining all the pieces

Once again to get it to work on your site you would only need to copy over the `gatsby-node.js` file located here.

And both the template and the search component.

Issuing `gatsby develop` again, and if all went without any issues one more time, open your browser of choice and enter the url `http://localhost:8000/search`, you'll have a fully functional search at your disposal coupled with Gatsby API.

Hopefully this rather extensive guide has shed some insights on how to implement client search using js-search.

Now go make something great!