A Software Engineering expertise search and navigation website

Andres Ponciano

May 2022

Contents

1 Abstract

Our website contains a curated list of research topics, scholars, and publications related to Software Engineering and Computer Science and its related topics. The goal was to make a website that is both easy to access and search through. As useful and vast as Google Scholar is, there is no dedicated site where people in the Software Engineering/Computer Science community can access data of Professors that focus on SWE/CS. Furthermore, Google Scholar has flaws such as not showing topics and making it somewhat hard to find specific profiles from the start. These design flaws paired with the fact that Google Scholar doesn't have a dedicated API that is open to the public allowed for the motivation of this project. We have written a website where scholars, their profiles, research topics, and publications are easily accessible and easy to find.

2 Introduction

There were two initial goals that we had in mind when we wanted to start writing our website. We knew we had to do data collection for the topics that we wanted and then allow for easy and effective navigation of this data using the most optimal and modern technologies available. In this section, we will explain the reasoning for every technology that we used.

2.1 Data Collecting

The first step to getting our website started was to collect the data that we wanted to offer. Initially, data was being collected using Python's Scholarly library. However, this library had limitations so we made a switch to use PuppeteerJS for fetching the data on publications. Lastly, we didn't want to reinvent the wheel or use antiquated methods when it came to searching so we used a RESTful search engine in ElasticSearch to allow for fast and effective searching of our data.

2.2 Backend

With ElasticSearch powering our search queries, we needed something that would pair well with it. GraphQL allows for querying from multiple sources so it seemed like a good choice to pair with ElasticSearch queries and any other methods that we might need in the future. We also had our GraphQL enpoint running on an Express server since it tied well with the technologies that we were using for our frontend.

2.3 Frontend

Apollo is one of the few communication clients that help manage flow of data between an applications client and backend that also works specifically with GraphQL. Furthermore, Apollo allows for query caching, which will speed up

```
search_query = scholarly.search_keyword(new_topic)
author = next(search_query)
print(author['name'])      You, seconds ago • Uncommitted changes
```

Figure 1: Code for fetching data using scholarly

any repeated searches. Lastly, React has been the leading library for frontend development in recent years and NextJS had been the most prominent framework because of its ease of use and allowing to do static and server-side rendering of HTML pages.

3 Data Collection

The approach for data collection was two-fold: the topics, people, and their information were fetched using Python's Scholarly pip package and the publications were fetched using Javascript's Puppeteer library since Google Scholar's site was blocking access from too many access calls. This was not an issue with Puppeteer. Afterwards, this data was indexed into an ElasticSearch instance so that we could easily fetch it.

3.1 Scholarly

Given a curated list of topics that cover the Software Engineering field, we fetched authors under those topics using scholarly's search_keyword method that returned an author's information as shown in Figure 1. From this data, we stored their names, scholar id's, pictures, affiliation and topics into an ndjson (newline delimited json) file for later use. We will later explain why storing it this way first was necessary.

3.2 Puppeteer

As mentioned before, Scholarly was not covering our needs when fetching for publications, so we had to switch our approach. Puppeteer served as a great alternative because we could control the speed at which we access Google Scholar and could therefore not get blocked by it. This is because Puppeteer is a Node library that can control a Chrome or Chromium browser. This means that we can simulate any type of web browsing using this tool using Javascript. With Puppeteer, we were able to crawl Google Scholar's website by using the functions provided. We did so by initiating a browser and a page in that browser like we would do ourselves except that done with code. With that page, we can assign it a URL as shown in Figure 2. In our case, we used command line arguments to provide the URL for each person. Once the web page is accessed through our browser, Puppeteer is able to access any part of a web page by inspecting for CSS selectors. Once these selectors are found within the page, we can perform

```
const browser = await puppeteer.launch({
    headless: false,
    slowMo: 200,
});

const page = await browser.newPage();
await page.goto(process.argv[2]);
```

Figure 2: Code for starting a browser with Puppeteer

actions on them like clicking, scrolling, typing, accessing text, and many more. This allowed us to click on every author's publications, copy the text of its title, abstract, and number of citations, and store it as an ndjson file as well. We also stored every person's total citations and h-index using this method.

3.3 ElasticSearch

ElasticSearch is a powerful search and analytics engine that allows for searching using simple REST API calls. ElasticSearch scales horizontally and is composed of "shards" which consist of inverted indices of documents and allows for full-text search. These indices each have their own documents the same way SQL has tables and rows. Each document follows a format called a mapping the same way SQL has schemas.

We have two indices: people and publications. Their mappings are in Figure 3 and Figure 4 respectively. We can see each of its data fields with their respective data types. It is worth noting that when a value is labeled as "text", ElasticSearch will tokenize to allow for token searching. We can see how a value can be tokenized when we run an analyze query as shown in Figure 5 and Figure 6. However, when these values are tokenized, we are no longer able to sort by their original values. As we can observe in Figure 3, our name field is labeled as "text" so that we can search by an person's tokenized name, but also has an extra field of type "keyword" so that we can also sort our results by name if we so choose to. Lastly, we also notice that "tag_clouds" is a field in the people's index even though we haven't fetched for it yet. This is because the tag clouds were built after fetching the publications. We used the titles and abstracts of every person's publications and calculated every word's term frequency and document frequency and used that metric to calculate the most relevant terms for every person.

At this point, we had all of our data was fetched, but it wasn't all that clean. Before we indexed our data into ElasticSearch, we performed some data cleaning on the publications using Python to remove duplicates and integrating all authors of the same publication into a single one. Only then were we ready to index out data for searching. This was a two-step process done using ElasticSearch's Python API. We first created an index by writing a mapping in JSON format and then calling the indices.create function. We can see the syntax on

Figure 3: Mapping for all people in Elasticsearch

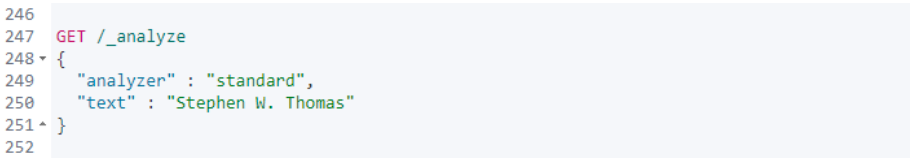
Figure 7. With the index for people and publications created, we could proceed to indexing documents to it. We simply stored all our publications in an array and called the bulk function as shown in Figure 8. Now the data was ready to be searched through, sorted, and fetched. At the time of writing this, we have 2,274 people and around 200,000 publications indexed, but we are working on increasing our data set.

4 Server

For our backend server, we have an Express server running on a GraphQL endpoint that is able to communicate with our ElasticSearch indices. Once, we are able to communicate with ElasticSearch, our GraphQL queries and resolvers - which we will describe later - are able to fetch and return our data in whatever manner we want.

```
{
  "publications" : {
    "mappings" : {
      "properties" : {
        "abstract" : {
          "type" : "text"
        },
        "num_citations" : {
          "type" : "double"
        },
        "pub_authors" : {
          "type" : "nested",
          "properties" : {
            "id" : {
              "type" : "long"
            },
            "name" : {
              "type" : "keyword"
            }
          }
        },
        "title" : {
          "type" : "text",
          "fields" : {
            "raw" : {
              "type" : "keyword"
            }
          }
        }
      }
    }
  }
}
```

Figure 4: Mapping for all publications in Elasticsearch



```
GET /_analyze
{
  "analyzer" : "standard",
  "text" : "Stephen W. Thomas"
}
```

Figure 5: Query to see how a field would be tokenized in standard mode

```
{
  "tokens" : [
    {
      "token" : "stephen",
      "start_offset" : 0,
      "end_offset" : 7,
      "type" : "<ALPHANUM>",
      "position" : 0
    },
    {
      "token" : "w",
      "start_offset" : 8,
      "end_offset" : 9,
      "type" : "<ALPHANUM>",
      "position" : 1
    },
    {
      "token" : "thomas",
      "start_offset" : 11,
      "end_offset" : 17,
      "type" : "<ALPHANUM>",
      "position" : 2
    }
  ]
}
```


Figure 6: Result for analyze query to see tokens

```
response = es.indices.create(
    index="people",
    body=mapping,
    ignore=400
)
```

Figure 7: Code for creating an index in Elasticsearch

```
helpers.bulk(es, people)
```

Figure 8: Code for bulk indexing data

```
type Person {
    id: ID!
    name: String!
    scholar_id: String
    url_picture: String
    affiliation: String
    topics: [String!]!
    other_topics: [String]!
    total_citations: Int
    h_index: Int
    tag_cloud: [String]
    highlight: [String]
}
```

Figure 9: Person Type in GraphQL

4.1 Express and Apollo

The basic boilerplate code for our Express server includes creating the Express app with its basic HTTP functionalities and applying a middleware as an Apollo/GraphQL server that contains our GraphQL graph. Our Express app then listens for the Apollo/GraphQL endpoint. This endpoint is initialized with the GraphQL schema that we will cover next.

4.2 GraphQL

As mentioned above, we have a GraphQL server which is initialized with two objects: typeDefs and resolvers. TypeDefs include all of our queries and objects types that can be used as output for a query. For example: query "person(id: Int): Person!" will take an id as input and return a Person type which is required as indicated by the exclamation mark. As seen in Figure 9 the Person type includes all the same fields that our mapping in ElasticSearch had. The resolvers object is the one responsible for handling the logic of each query. Here is also where we communicate with our ElasticSearch client. For every query, there is a resolver and in every resolver, we will have a schema that contains the ElasticSearch query. These queries are in a JSON object format and can be passed into an ElasticSearch client to fetch the data requested. The ElasticSearch schema for the person query can be seen in Figure 10 and we call the ElasticSearch client as shown in Figure 11. Whether we are sorting or doing pagination, every resolver for every query follows the same logic: figure out the ElasticSearch query schema needed, pass it on the client and return the data.

5 Client

As mentioned before React has become in not-so-recent years the leading tool used for frontend development. React allows for easy modularization of web page components and controlling a pages state in different ways. NextJS only exacerbates React's upsides and adds onto it. Furthermore, we used to Apollo's

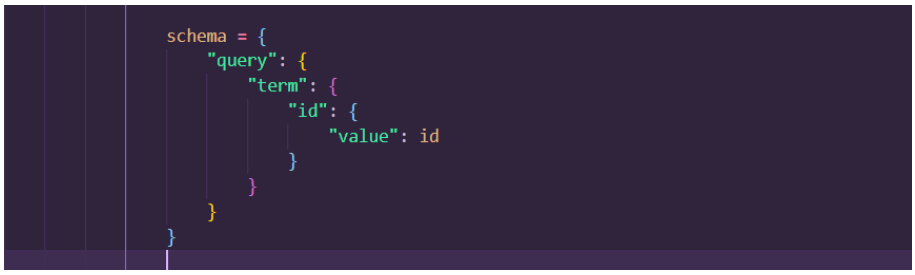


Figure 10: JSON schema for searching for a person given their ID in ElasticSearch

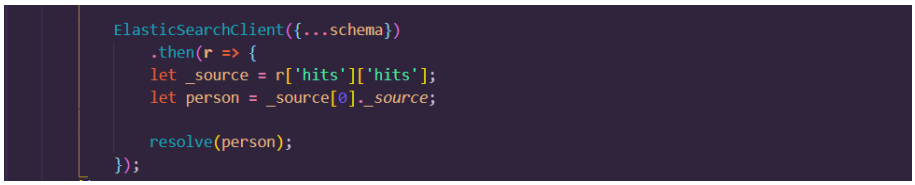


Figure 11: Calling ES client in GraphQL resolver

React client library allows us to communicate with our backend. Lastly, Tailwind is the cherry on top that styles all of our components alongside HeadlessUI to perform quick styling and creating modern UI components such as dropdowns, listboxes, and menus.

5.1 NextJS

Although React already allows for modularization of components, NextJS takes it a step further by dividing things into pages and components. These pages are written the same way as any React component except that NextJS only allows for its server-side and static rendering methods on the pages and not in the components and NextJS automatically creates routes on pages without the need for any other router methods. Because of this, our website has three main pages - the home page, the people's page, and the publications page. Each of these pages has its own data fetching and components that it uses like dropdowns, search bars, lists, etc... Furthermore, we needed a page for every person's profile page. This was done using dynamic routing which is also enabled by default with NextJS as long we structure our file structure the right way. Wrapping all of these pages is our ApolloProvider and Layout. The layout is simply anything that we want to display on all pages of our website. In our case, this was only a navbar. The ApolloProvider will be discussed in the next section.

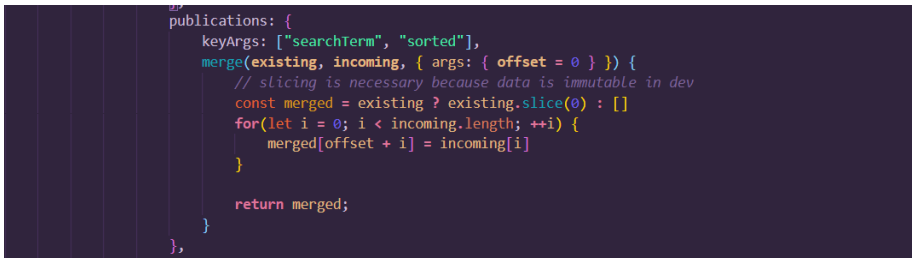


Figure 12: caching policy for publications queries

5.2 Apollo Client

Like we have mentioned before, Apollo connects our backend with our frontend. Although this is the main use of the Apollo client - and we will show how we call our backend later - we also use the client to specify our query caching policies. Every caching policy can depend on a few things: what parameter we use to fetch the query, the id's of objects, and if or how we're doing pagination. For example, the publications caching policy, as seen in Figure 12, has a keyArgs array that says to cache a result based on both the "searchTerm" and the "sorted" variable of a query; The merge function of the same policy allows us to call a function that returns the previous results of a query plus some new results fetched of the same query. This allowed us to make use of the Apollo caching capabilities to make repeated queries faster while keeping the data that we are displaying consistent to what we want to show the user.

5.3 Home, People, and Publications

The three main pages are of similar structure so we will describe them together. To oversimplify, there are three things that make up each of these pages.

1. A GraphQL query called using Apollo's useQuery or useLazyQuery function.
2. Pieces of state managed and updated using React's useState hook sometimes accompanied by useEffect.
3. A combination of Javascript and HTML - known as JSX - to render and handle any action, condition, function, etc...

As we have seen in the backend section, a GraphQL query has a resolver that will fetch data from our ElasticSearch indices and return it to the client. In the frontend, we call those queries using a function called useQuery or useLazyQuery that the Apollo module provides. We will explain the difference between these two later, but they both work almost the same; They will both return a loading, error, and data object, have a fetchMore function (this function would be the

```
const SEARCH_PUBLICATIONS_QUERY = gql`
  Execute Query
  query NewPubsPag( $searchTerm: String, $offset: Int, $limit: Int, $sorted: String ) {
      publications( searchTerm: $searchTerm, offset: $offset, limit: $limit, sorted: $sorted ) {
          title
          abstract
          num_citations
          pub_authors {
              id
              name
          }
      }
  }
  `;
```

Figure 13: Example of graphQL query

```
const { loading: loadingSearch, error: errorSearch, data: dataSearch, fetchMore } = useQuery(SEARCH_PUBLICATIONS_QUERY, {
    variables: { searchTerm: homeSearchValue, sorted: sortedBy.value }
});
```

Figure 14: Example of useQuery function

one to call the caching policy discussed earlier), and both can take in a variables object to pass into the GraphQL query. The only difference between them is that useQuery will only run once when a page renders, and useLazyQuery will run when called from a specific action (like pressing a button). Ultimately, they both do what we want to do with our backend - retrieve data. An example of a GraphQL query and its respective useQuery call is show in Figure 13 and Figure 14. Furthermore, in Figure 15, the syntax for useLazyQuery is slightly different as it has an attribute called "getSuggestedResults". This is what is called when we want to fetch the data using this query since it can only be fetched through an action.

These functions also allowed us to do pagination in two different ways: offset and cursor pagination. Offset pagination was done by passing in two variables: offset and limit. To explain simply, if we wanted page 2 of the people in our database, we would pass 10 as the offset and 10 as the limit. This means that we return the results starting at offset 10 and we only return 10 results per page. To access the next page, we would simply pass offset as 20 and the same limit. Cursor pagination works with the fetchMore function because of the cache policy we described earlier. We fetch a certain amount of results and when we call the fetchMore function, we will return the previous results plus 10 more. This is possible and inexpensive because of the previous results are cached and the merge function in our cache policy simply adds to these previous results.

If you want to keep track of something in a web page when coding with React, you want to do it with state. There are other ways of tracking state with React, but with our current setup useState was sufficient. With useState we: tracked user input for our searches, the topic to be searched for, the current sorting

```
const [ getSuggestedPeople, { loading: loadingSugg, error: errorSugg, data: dataSugg } ] = useLazyQuery(SUGGESTED_PROFILES_QU
  variables: { prefix: searchText }
});
```

Figure 15: Example of useLazyQuery function

option, and anything else that would require us to keep track of it. UseEffect can be used to track page rendering (depending on what conditions you set for it as well). In React, a page re-renders when a piece of state changes (it is important to note that only the parts of the page that change are updated) which is why these two work in tandem. For example, when tracking the topic to search for we create a piece of state by writing "const [topic, setTopic] = useState("")". This will initiate our "topic" state with a function "setTopic" that can update the state and has an initial value of an empty string. When our state changes through the setTopic function, our page will re-render and we will fetch new results - now filtered by the topic. In other cases, a useEffect function will be used to modify the workflow of things.

JSX looks similar to what you might imagine if you are familiar with HTML and Javascript. Every page/component is a Javascript arrow function that can take in parameters and is exported at the end. You can write Javascript functions, variables, and if statements before a page/component renders. Finally, what you return is a piece of HTML code that can include anything from divs, anchor links, paragraphs, and anything else you can do with vanilla HTML plus the addition of being able to interact with the Javascript code that you wrote before the HTML code. This is the basis of React and is used all throughout our webpage.

5.3.1 Searching

Our application allows for searching of people's names, and their keywords (tag_cloud terms) and publications titles and abstracts. We also allow for filtering by a person's topic, and sorting by name and sorting publications by their title and number of citations. As explained earlier, we tracked the input a user typed into our search bar using useState. As the user types our state changes which means our page re-renders. When we re-render the page, we have a useEffect function that will trigger a useLazyQuery for suggested results to display under our search bar. Once a user decides performs an action to search, we will again re-render our component so that our useQuery fetches the data we want to display. Finally, our JSX is setup in a way that the UI displays the data fetched, highlights the terms we searched for, allows for navigation through pagination, and has everything necessary for the user to perform a different search.

5.4 People's Profiles

A person's profile will include all the same items described in the previous section. We differentiate this page from the others for one reason - it is implemented using dynamic routing. Dynamic routing simply means that an HTTP route can take variables, therefore making it dynamic and not static. This also means that we can use NextJS to our advantage and pre-render each of these pages by simply including a getStaticPaths function in our page. Data fetching, pagination, and displaying the data works more or less the same as the other pages we described before.

5.5 Tailwind and HeadlessUI

CSS is the cherry on top of every web page by styling and commanding how the UI should be organized. In our case, we used the Tailwind framework to simplify the way in which we integrate CSS into our HTML components. This is done by writing the styling directly inside the HTML components in the form of their class names. Furthermore, HeadlessUI was used to implement modern UI components like dropdowns while allowing us to fully style them with Tailwind.

6 Conclusion

With the help or modern technologies, we successfully fulfilled the two goals we had set for this project: we collected the necessary data and then we effectively fetched and displayed the data. The biggest challenge was working around the limitations that Google Scholar sets on data collection by blocking access when Google Scholar is used too much in too short of a time frame. We were able to bypass this challenge with the help of Puppeteer. The next biggest challenge was being able to use ElasticSearch along with the other technologies that we wanted to use. As we finalized the website, it became clearer how useful this tool can be especially when comparing it to how Google Scholar has set their website up. It is true that Google Scholar allows us to fetch for about the same data as our site, but Google Scholar does not have in mind users that would want to fetch for specific topics, users, or areas of study. Our website takes into consideration users that would be trying to search for these things and we try to make it as easy as possible for them to find what they need. The code to everything that was covered in this report can be found in my Github profile at this repository(https://github.com/AndresPonciano/finalprojectforsure).