Proposal for GSoC 2023

Open API Web Search

- **Name:** Priyanshu Sharma
- **Country of Residence:** India
- **Telephone:** +918192046574
- **Timezone:** IST (India)
- **Typical Working Hours:** 6:00 am to 10:00 am, 11:00 am to 2:00 pm, 3:00 pm to 6:00 pm(IST) -> 2:00 am to 6:30 am, 7:30 am to 10:30 am, 11:30 am to 2:30 pm(UTC)
- **Mail:** priyanshuSharma507@proton.me
- **Personal Website:** https://priyanshu-kun.netlify.app
- **Github:** https://github.com/priyanshu-kun
- **Language:** English, Hindi

Why I've chosen this project & postman as my Organization.

Believe me, I always have a "why" before taking any significant actions, and I have a very clear reason and motivation for this particular project and selecting Postman my organization. I admire the development of the Postman, which started out as a straightforward HTTP client meant to address a personal issue and has since grown into a complete organization used by over 20 million developers across the world. Sometime back, I came to know about Postman **The API First World** vision that caught my interest. Within that vision, developers build faster and better software with three main pillars.

- APIs are considered the #1 priority in the software development lifecycle.
- APIs are easily consumable via end users.
- APIs that are easily discoverable. ***(Goal)***

In API first world **100 million developers are connected through APIs**, and APIs take center stage as primary building blocks.
I highly think this project could be part of **APIs that are easily discoverable** pillar. It could serve as a unified platform to get various kinds of valid API definitions, repositories, and proper documentation for all developers across the globe by utilizing this project, we can ensure that APIs are easily discoverable. If I were to somehow get the chance to work on this, it would give me a purpose to work on something meaningful.

Project Description -

Name: Open API Web Search.
Mentor: @vinitshahdeo, @MikeRalphson
Goal: The goal is fairly straightforward: I need to develop a solution for extracting Swagger and OpenAPI definitions from less-known sources on the open web and common crawl. The solution must crawl through the common crawl Index files to search for API definitions, validate them, and indexing them which can be done by Elastic Search, which provides a simple and effective interface as a part of an easy search. **The TLDR; is the solution that should function like a search engine for OpenAPI definitions**.

Technical Knowledge -

I am a 2022 graduate student from Sri Dev Suman Uttarakhand Vishwavidhalaya. I am enrolled in a 3 years BCA(Bachelor of Computer Applications) course. My course mainly focuses on Computers and their applications (it's more of discrete maths and computer applications though). The courses that I have done include

- C/C++ Programming
- Discrete Mathematics
- Computer graphics
- Computer Architecture
- Java Programming
- Data Structure and Algorithms
- Operating Systems
- DBMS
- Computer Networks
- Web Technologies
- E-Commerce

I believe that this project heavily relies on the development and web skills, which happens to be my area of expertise. I possess good development and web skills, having completed two medium to large projects and deployed them on the cloud.

I have done many Udemy courses such as

- Web Development Bootcamp by Dr. Angela Yu
- Nodejs Bootcamp by Andrew Mead and Rob Percival
- ReactJS Bootcamp by Colte Steele

Prior experience in Web Development:

- **Kira** - Kira is a Full-Stack web-based application that allows developers to easily track, manage and collaborate with other developers on bugs in their software projects.
- **codersHouse -** A Full-Stack web-based Application online audio chat web application is a platform that allows users to communicate with each other in real-time via audio

- Amazonia - A full-stack Amazon clone that includes user authentication and authorization features, as well as the ability to handle payments through both Stripe and PayPal.

Plan & technical details -

Plan

- **Steps:**
  - **Crawling:** The task at hand involves utilizing web scraping to obtain OpenAPI definitions from the open web and Common Crawl through the implementation of Node.js. Specifically, the process will entail scraping data and subsequently storing it in an Elasticsearch database for indexing.

  - **Validating:** Now that we have our Swagger and definitions, we need to validate their responses before indexing them. Definitions validation can be done using the **swagger-parser** library.

  - **Indexing:** This part can be done by Elasticsearch. The openAPI definations data can be indexed using elasticsearch, enabling quick and effective search searches. Elasticsearch can manage various data categories, including text, numeric, and geographic data, and can automatically build and refresh the index based on the data structure.

  - **Implementing a search algorithm:** A variety of algorithms and search techniques can be used, and Elasticsearch can be used as the tool to accomplish that it provides a powerful search API that supports various search algorithms, such as full-text search, fuzzy search, geospatial search, autocomplete search and personalized search. The search API also supports advanced features such as faceting, filtering, and sorting, which can improve search efficiency and user experience.

  - **Providing an interface:** Implementing a UI Interface for Search functionally. That can be done using Nextjs! Why not create UI using Reactjs? Because Next.js provides many benefits over building with React alone, including server-side rendering, automatic code splitting, easy setup and configuration, built-in routing, and static site generation.

These features can help build faster, more performant, and more scalable user interfaces.

    - **Updating dataset:** That completes the trip we are on! To improve searching, we must frequently refresh the crawl results and reindex the Elasticsearch information. By setting up a cron task, we can crawl data in the background and index it as we discover definations.

- **Technologies and Tools:**
    - Next.js, Node.js, Elasticsearch, Express.js, Figma.
- **Support I need from the community:**
    - Discuss with mentors in case of query.

Technical Details

API Endpoint:

POST /api/search	Query(keywords/term, sort_order, page_no, page_size, user_id);

Here is a functional dummy example of the project:

I developed a mock OpenAPI web search project as part of my proposal to test my strategy, and the code I made for my proposal includes a thorough description of the app. You can learn more about my skills and the development process I use by looking at this Dummy App. I used the following tools to create the front-end and back-end functions of the app:-

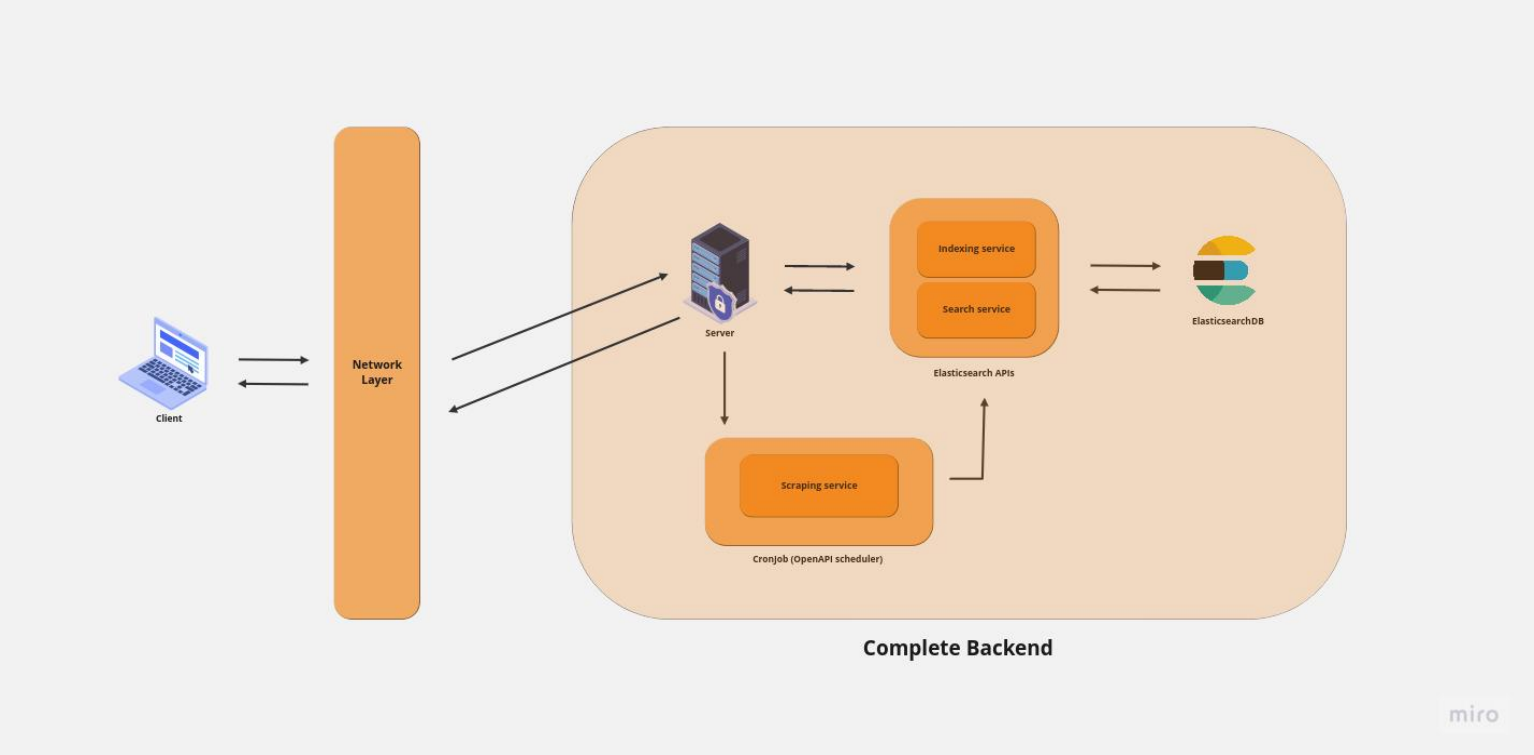
- **Frontend**
    - Vanilla javascript
    - HTML5
    - CSS
    - vite

- **Backend**
    - Expressjs
    - Elasticsearch
    - Kibana
    - cheerio
    - node-cron
    - swagger-parser

URLs:

Live	Source Code
https://openapi-web-search.netlify.app	https://github.com/priyanshu-kun/openapi-web-search

The core Application design and dataflow are shown below.



Below I have divided our solution into the following services.

Services:

- **Crawling Service:**

  We can start by scraping the index file URLs from the common crawl index server. To accomplish this, we can use Node.js' **cheerio** module, which can retrieve the files which contain the URLs for index files on the internet. We can use **axios** to follow these URLs and retrieve the associated index files once we have them. These files may be quite large, and we must conduct a line-by-line

scan to retrieve their information. We can accomplish this by utilizing the Node.js **fs** and **readline** tools.

To begin the scraping procedure, submit a GET request to the given URL. We will receive an HTML document if the call is successful. The next step is to collect all anchor tags within the table using **cheerio** that links to a file holding the route to the index files on the internet.

COMMON_CRAWL_SERVER_URL: `https://index.commoncrawl.org/`

```
async getDataFromCommomCrawlServer(url) {
      try {
          const links = [];
          const { data } = await axios.get(url);
          const html = data;
          const $ = await cheerio.load(html);
          $('tbody tr').each((index, row) => {
              const linkTd = $(row).find('td:last-child');
              const link = linkTd.find('a').attr('href');
              links.push(link)
          });
          return links;
      }
      catch (e) {
          // handle errors gracefully
      }
  }
```

Expected Output:

```
'https://data.commoncrawl.org/crawl-data/CC-MAIN-2023-06/cc-index.paths.gz'
'https://data.commoncrawl.org/crawl-data/CC-MAIN-2022-49/cc-index.paths.gz'
'https://data.commoncrawl.org/crawl-data/CC-MAIN-2022-40/cc-index.paths.gz'
'https://data.commoncrawl.org/crawl-data/CC-MAIN-2022-33/cc-index.paths.gz'
'https://data.commoncrawl.org/crawl-data/CC-MAIN-2022-27/cc-index.paths.gz'
.....
```

The code below is a javascript function it sends a GET request to the fetched URLs and reads the response in buffers. Since the response is compressed in .gz format, we must decompress it before reading it. To accomplish this, we

are utilizing the **zlib** module in Node.js. Once we have decompressed the response, we can use the **readline** module to read the content of the file line by line.
The output of the code snippet will be the path for the index files located on the internet. To access the data, we need to prefix this path with the following URL: `https://data.commoncrawl.org/`.
To read the data from index files, it is necessary to first download them onto the machine using the following code:

```
async downloadFile(url, path) {
const response = await axios({
            url,
            method: 'GET',
            responseType: 'stream',
            onDownloadProgress: function (progressEvent) {
                const total = progressEvent.total;
                const downloaded = progressEvent.loaded;

                const percent = Math.round((downloaded / total) * 100);

                process.stdout.clearLine();
                process.stdout.cursorTo(0);
                process.stdout.write(`Downloading: ${percent}%
(${downloaded}/${total} bytes)`);
            }
        });

        const fileStream = fs.createWriteStream(path);
        response.data.pipe(zlib.createGunzip()).pipe(fileStream);

        return new Promise((resolve, reject) => {
            fileStream.on('close', () => {
                resolve();
            });

            fileStream.on('error', (err) => {
                reject(err);
            });
        });
    }
```

Once the file has been downloaded, we need to read its content in 1MB chunks. While reading the textual content of the file, we also need to parse it.

```
                const readStream = fs
                .createReadStream('./data/output.txt',
                 { highWaterMark: CHUNK_SIZE });
                readStream.on('data', chunk => {
                    const text = chunk.toString();
                    const parsedUrls = parsing(text)
                    if(parsedUrls !== undefined) {
                        console.log(parsedUrls)
                    }
                });
                readStream.on('end', () => {

            // once reading file is completed so we don't need it
            // we can safely delete that
            // if deletion process is successfully completed then we
            // call the code recursively to read next file

                });
                readStream.on('error', err => {
                    console.error(err);
                    return;
                });
```

After completing the reading, the file is no longer needed and can be removed from the system to read the next text file.

```
async removeFile(url) {
       try {
           fs.unlink(url, (err) => {
               if (err) throw err;
               console.log('File deleted!');
             });
       }
       catch(e) {
           console.log(e)
```

```
        }
    }
```

Recursively reading next file:

```
readStream.on('end', () => {
                    removeFile("./data/output.txt").then((res) => {
                        const flattedUrlsForIndexing =
urlsForIndexing.flat()
bulkIndexOpenAPIs(client,flattedUrlsForIndexing).then(res => {
                            readContent(client,links, idx + 1);
                        })
                        .catch(e => {
                            console.error('Error while Indexing: ',e);
                            return;
                        })
                    }).catch(e => {
                        console.error('Error while deleting file:', e)
                        return;
                    })
                });
```

Complete Implementation:

```
function readContent(links, idx) {
    if (idx < links.length) {
        downloadFile(links[idx], './data/output.txt')
            .then(async () => {
                const readStream = fs
                .createReadStream('./data/output.txt',
                 { highWaterMark: CHUNK_SIZE });
                readStream.on('data', chunk => {
                    const text = chunk.toString();
                    const parsedUrls = parsing(text)
                    if(parsedUrls !== undefined) {
                        urlsForIndexing.push(parsedUrls)
                    }
                });
```

```
                readStream.on('end', () => {
                    removeFile("./data/output.txt").then((res) => {
                        const flattedUrlsForIndexing =
urlsForIndexing.flat()
bulkIndexOpenAPIs(client,flattedUrlsForIndexing).then(res => {
                            readContent(client,links, idx + 1);
                        })
                        .catch(e => {
                            console.error('Error while Indexing: ',e);
                            return;
                        })
                    }).catch(e => {
                        console.error('Error while deleting file:', e)
                        return;
                    })
                });


                readStream.on('error', err => {
                    console.error(err);
                    return;
                });
            })
            .catch((err) => {
                console.error('Error downloading file:', err)
                return;
            });
    }

}
```

Incorporating parallel processing in our Solution -

As we all know, processing large files can be a time-consuming and resource-intensive job, as I demonstrated above, and I think we can make significant improvements by incorporating the **child_process or cluster** module into our application. We can use multi-core systems to handle the file in parallel by using the **child_process or cluster** module. This means that we

can spawn multiple child processes to download and parse various sections of the file at the same time, greatly reducing processing time. This is particularly useful when dealing with large files or when working with limited resources. Using the **child_process or cluster** module can make our program more reliable and fault-tolerant in addition to improving performance. We can ensure that any failures or errors in one process do not affect the others by isolating the downloading and parsing tasks into separate processes, and we can more easily debug and troubleshoot any issues that emerge.

Overall, I think that incorporating the **child_process or cluster** module into our application will improve its efficiency, scalability, and reliability, as well as allow us to better manage Index files. I'm sorry, I'm not able to provide you with a code example at this time as this is a last-minute change.

Parsing:
To locate swagger and openapi definitions, it is necessary to parse the content of each line when reading through an index file cause index files can be large. Regular expressions can be utilized to extract the content from the files. It is important to extract the URLs from the index files using specific keywords when employing regular expressions.

- ➔ **openapi.json**
- ➔ **openapi.yaml**
- ➔ **swagger.yaml**
- ➔ **openapi.yml**
- ➔ **swagger.yml**
- ➔ **swagger.json**
- ➔ **swagger**
- ➔ **openapi**
- ➔ **api subdomain**

I have created the following regular expression for this task, it extracts all URLs that contain those keywords in their respective paths.

Regex:
Regex for extracting URL from the text:

```
/http[s]?:\/\/(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+/gi
```

Regex for check if a URL contains “openapi” and “swagger” as keywords

```
/^(?:https?:\/\/)?[^\/\s]+(?:\/[^\/\s]+)*(?:\/(openapi|swagger))(?:\/[^\/\s]*)*$/gi
```

Regex for checking “api” as subdomain.

```
/^(?:https?:\/\/)?api\.[^\/\s]+\.[^\/\s]+(?:\/[^\/\s]*)*$/gi
```

Regex for checking files definitions files.

```
/^(openapi|swagger)\.(json|yaml|yml)(\?[\w=&]+)?$/gi
```

Code:

```
parsing(text) {
       const urlRegex =
/http[s]?:\/\/(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+/gi;
       const matchUrls = text.match(urlRegex)
       const parsedUrls = [];
       if (matchUrls) {
           const keywordReg =
/^(?:https?:\/\/)?[^\/\s]+(?:\/[^\/\s]+)*(?:\/(openapi|swagger))(?:\/[^\/\s]*)*$/gi;
           const subDomainReg =
/^(?:https?:\/\/)?api\.[^\/\s]+\.[^\/\s]+(?:\/[^\/\s]*)*$/gi;
           matchUrls.forEach(url => {
               const tokens = url.split('/');
               const ext = tokens[tokens.length - 1];
               const rg =
/^(openapi|swagger)\.(json|yaml|yml)(\?[\w=&]+)?$/gi
               if (keywordReg.test(url) || subDomainReg.test(url) ||
rg.test(ext)) {
                   parsedUrls.push(url)
               }
           })
       }
       if(parsedUrls.length !== 0)
       return parsedUrls;
   }
```

Explanation:
The function takes a string input `text` and uses a regular expression `urlRegex` to extract URLs from it. The `match()` method is used to find all URLs that match the defined regular expression, and these URLs are stored in the `matchUrls` variable.

Next, an empty array `parsedUrls` is initialized to store the parsed URLs that meet the defined criteria. The function checks if any URLs were found using the `matchUrls` variable.

Two additional regular expressions, `keywordReg` and `subDomainReg`, are defined to match URLs containing either "openapi" or "swagger" as a directory and URLs containing the string "api." as a subdomain, respectively.

The function iterates through each URL found by the `matchUrls` variable using a `forEach()` loop. The URL is split into an array of tokens separated by slashes, and the last token is extracted as the file extension using `tokens[tokens.length - 1]`.

Another regular expression `rg` is defined to match files with the extensions .json, .yaml, or .yml and the keywords "openapi" or "swagger" in the file name.

The function tests whether the URL matches any of the three defined regular expressions using the `test()` method. If the URL meets any of the criteria, it is added to the `parsedUrls` array using the `push()` method.

Finally, the function checks whether any URLs were added to the `parsedUrls` array and returns the array if it is not empty.

- **Indexing Service:**

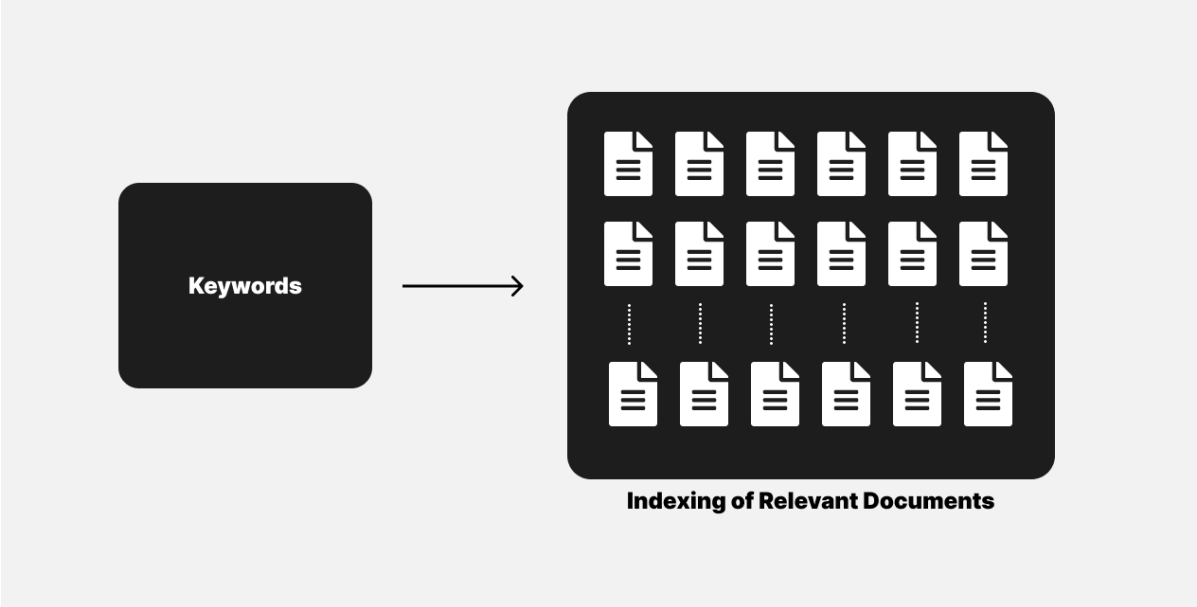
Let's see What is Indexing?
Documents are much easier to find when we grouped them in a logical manner the documents that share similar traits and are logically related to each other are grouped together called indexing.
Elasticsearch is meant to do that, it can accomplish document retrieval with extremely low latency. It is because instead of searching text directly, it searches in the index instead.

Elasticsearch has something called an **Inverted Index!**
Let's dissect the Algorithm which is followed for creating the index. First of all, Elasticsearch is provided a set of all documents and it will tokenize the contents of each and every document tokenize meaning it will split the text into individual words then it will create a unique set of all these unique words which will be in sorted order so there will be no repetition of terms only unique terms will be in Set and many stopped words might have been removed from this unique Set. After that, with each term, we will associate a list of documents in which those terms are found.

For eg:
Suppose we have two Docs.
Doc1: I am learning the cool stuff
Doc2: I am learning to learn

```
Am -> [Doc1,Doc2]
I -> [Doc1,Doc2]
Cool -> [Doc1]
Learn -> [Doc1,Doc2] // root for the learning
The -> [Doc1]
Stuff -> [Doc1]
```

Diagrammatic view of indexing inside Elasticsearch.



So, my approach for indexing is we will start getting the OpenAPI documents and parsing them. Elasticsearch will tokenize the content of files and index them with their corresponding documents.

This is how the indexing process in Elasticsearch takes place.

Solution walkthrough:
We can now begin crawling our URLs for search purposes, but first, we must extract the OpenAPI definitions from those URLs.

For eg:

```
const response = await axios.get(url);
```

and then index that information using swagger-parser for search Algorithm.

For eg:

```
const document = await SwaggerParser.parse(response.data);
```

Because the number of URLs could be quite big, we must mass index them, which can be accomplished using the `client.helpers.bulk();` technique in Elasticsearch. Here is an illustration of my indexing approach in its entirety.

```
async bulkIndexOpenAPIs(client, urls) {
      // Create an array of bulk index actions for each OpenAPI document
      const bulkActions = [];
```

```
    for (const url of urls) {
        try {
            const response = await axios.get(url);
            const tokens = url.split('/');
            const ext = tokens[tokens.length - 1];
            const rg =
/^(openapi|swagger)\.(json|yaml|yml)(\?[\w=&]+)?$/
            let document = null;
            if(rg.test(ext)) {
                document = await SwaggerParser.parse(response.data);
            }

            bulkActions.push({
                index: { _index: 'openapi_docs' },
                body: {
                    url,
                    content: response.data,
                    contentType: response.headers['content-type'],
                    info:  document?.info,
                    description: document?.description,
                    tags: document?.tags
                }
            });
            console.log("Done: ",url)
        }
        catch (e) {
            console.log("Address not found!");
        }
    }

    const result = await client.helpers.bulk({
        datasource: bulkActions,
        onDocument() {
            return {
                index: { _index: 'openapi_docs' }
            }
        }
    });
    console.log(result)
}
```

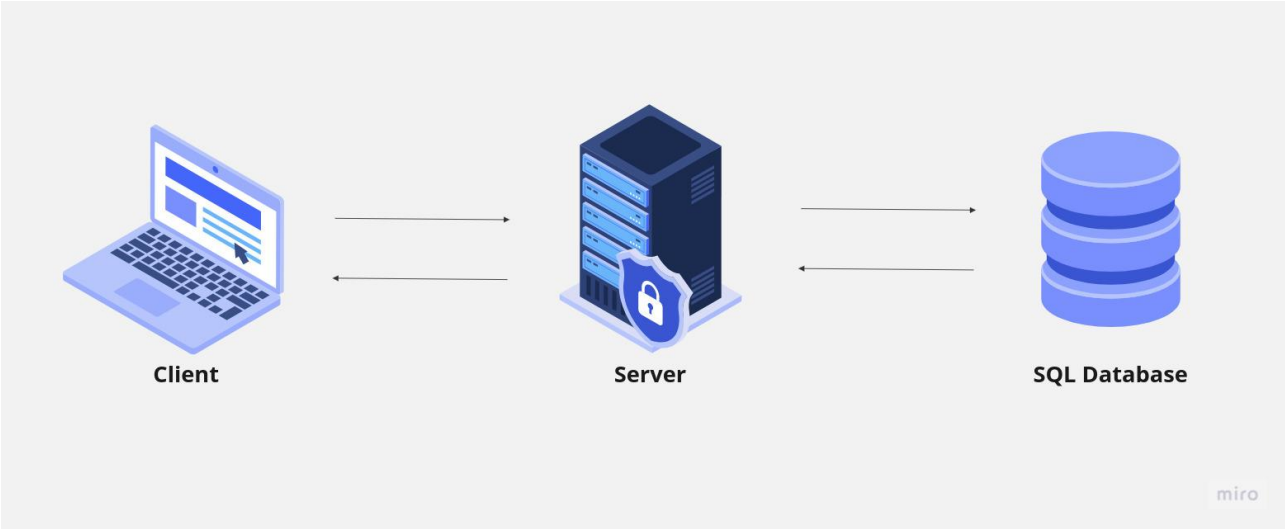
- **Implementing Search Service:**

  Search is an experience. This is the most crucial step of our approach, Implementing a search algorithm for blazingly fast retrieval/search is the first requirement for a search engine and Elasticsearch is meant for doing that. Whether we were searching for documentation or your favorite show and in that case openAPI documents you expect fast and relevant results no matter the scale.

  let's discuss how elastic search allows you to get fast search results regardless of scale.

  But First Why Elasticsearch?

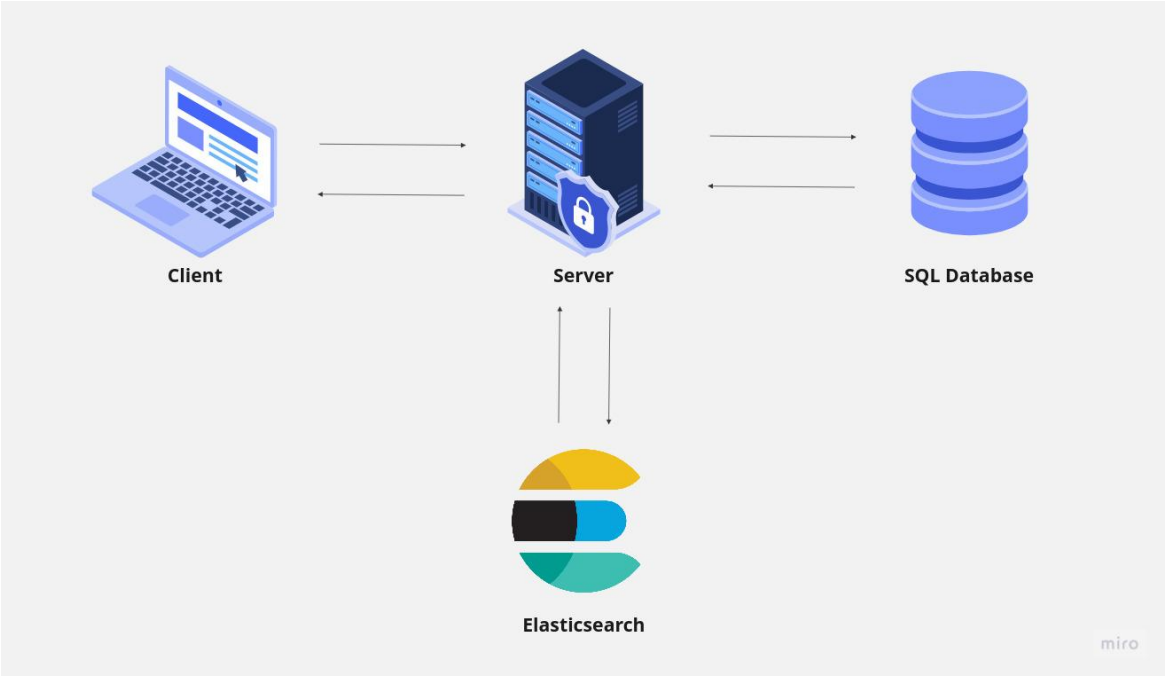
  Now, Imaging you are a lead developer responsible for building and maintaining a search engine application, and using that search bar millions of users search for the document they need so currently you have a full stack app connected to a SQL database now the database contains all webpages along with other data which is collecting from users and the internet. When a user search for something on our app the request is sent to the server which in turn looks for pages within the database and the response is sent back to the client so you can render it to the user browser.



  A great search experience is key to retaining users and we want the users to get fast and relevant results no matter the scale. So with the current setup that, I showed above the chances are that we may be working on with huge

dataset that is stored in our relational database and the data is scattered in multiple tables, and fetching the data from that tables could cause the performing issues in getting the search results.
And that could be a huge turn-off.
The important factor in search experience is relevancy, the whole point of search is finding relevant data fast and we want to be able to set criteria to have the most relevant result at the top and the least at the bottom even if the user misspelled the sentence or words.
And databases are not equipped to handle all of that so if you are in a situation where the speed and relevance of your search is an important aspect of your work Elasticsearch could come in handy.

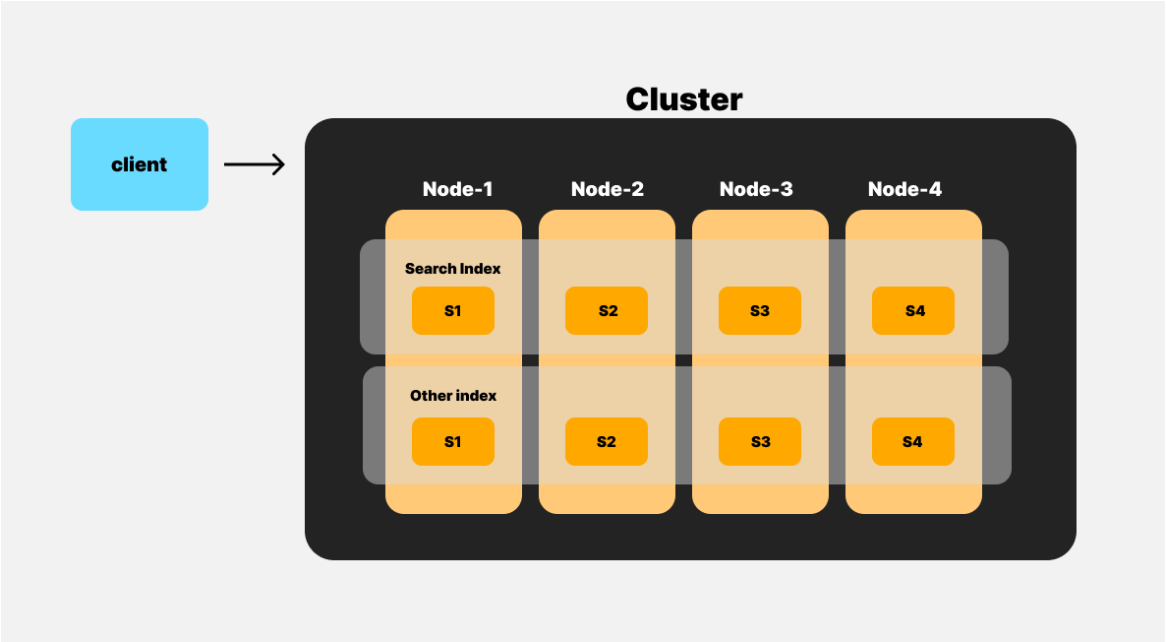
So, how does it look when Elasticsearch is connected to your app?



When a user sent a search query to your app the request is sent to the server which sent the search query to Elasticsearch. Now, Elasticsearch is meant to send relevant results fast to the server which processes the info and sends it back to the client.

The Architecture of Elasticsearch.

Elasticsearch is a powerful search and analytic engine which is known for its distributed nature speed and scalability, this is due to its unique architecture. The basic unit of Elasticsearch is a Node, it is an instance of Elasticsearch and now each node has a unique id and name and it belongs to a single cluster. When we start up a node a cluster is formed automatically and you could have one to many nodes in a cluster these nodes are distributed across separate machines but they all belong to the same cluster and works together to accomplish a task.
A node can be assigned one or multiple roles and one of the roles that a node could be assigned is to hold data.
There is a unit inside a node called a shard, the shard is the place at the Disk where the actual data is stored, and this is where we run a search. Indexing is also done by using shards when you create an index one shard comes by default, shard comes with a lot of superpowers you can configure it so you can create an index with multiple shards that are distributed across nodes, and that is the process called sharding.



Sharding is very important for horizontal scaling it can be done by distributing the data into multiple nodes.

In the above text, I'm so much talking about relevance, Before I'm talking about the actual search algorithm, let's talk about relevance first cause the whole search algorithm is all about relevance.
Suppose, you are searching for something on the internet but you're not quite getting what you are looking for and that is what relevance is all about.
When you are searching on an app you want results that are directly related to what you are searching for.
So, which brings us to the question of how we measure the relevance of our search results now the two factors that we will focus on are:

- **Precision**
- **Recall**

Precision and Recall are used to measure the relevance of a search engine.
When we send a search query to Elasticsearch it retrieves documents that are considered relevant to the search query.

There are a couple of terms that should be taken into consideration.
True Positives: These are the relevant document that is returned to the user.
False Positives: These are the irrelevant document that is returned to the user.
True Negatives: These are the irrelevant document that is not returned to the user.
False Negatives: These are the relevant document that is not returned to the user.

The Term Precision is defined from the given Formula:

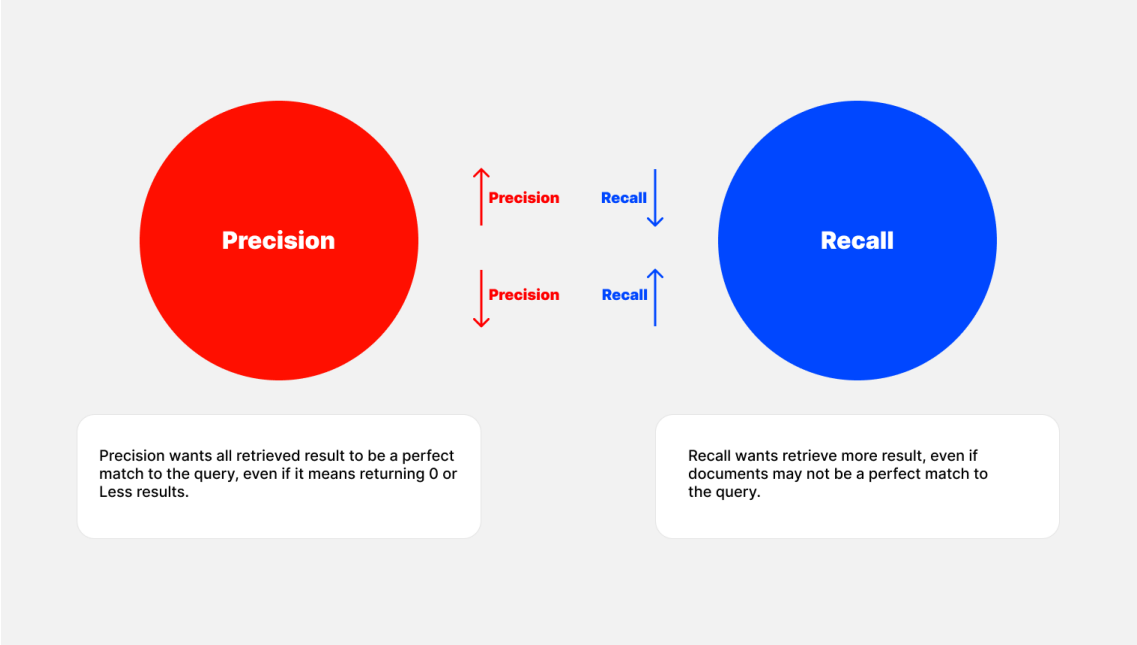
```
Precision = True Positives / True Positive + False Positive
```

So what precision tells us is which portion of retrieval data is actually relevant to our search query.

And, Recall on the other hand is calculated by:

```
Recall = True Positives / True Positive + False Negatives
```

What recall tells us is which portion of relevant data is being returned as a search result.

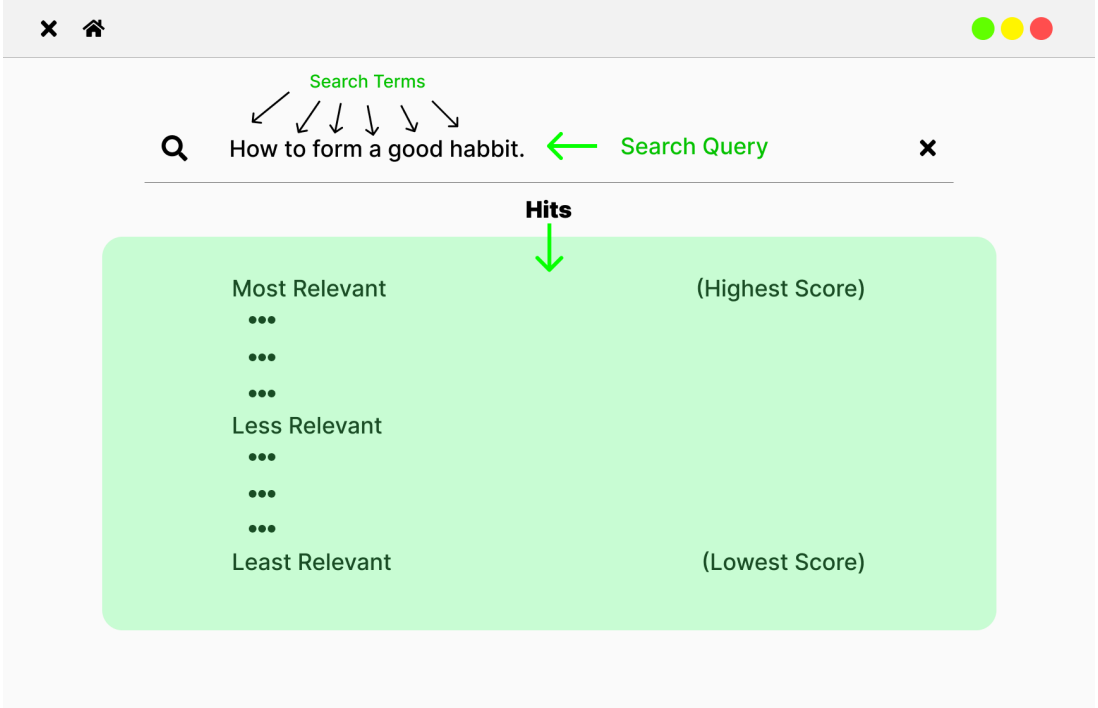


Precision and Recall determine which documents are included in search results but do not determine which of the returned document are more relevant compared to others!
This is determined by ranking, so when you look at your search results you will see the most relevant results are at the top and the least relevant at the bottom. And this ranking or order is determined by a scoring algorithm so each result is given a score and once with the highest score is displayed at the top, whereas the lowest score is at the bottom.

What is the score?

- The score is a value that represented how relevant a document is to that specific query.
- A score is computed for each document that is hit and hits are search results that are sent to the user.
- Term Frequency (TF)
    - Term Frequency determines how many time each search term appear in a document so if a document mentions a search term more frequently Elasticsearch assume that this document is more relevant to the search query it assigns a higher score to than document.
- Inverse Document Frequency (IDF)

- IDF diminished the weight of the term that occurs very frequently in the document set and increase the weight of terms that occur rarely.



The Algorithm!

One of the most commonly used search algorithms for full-text search is the BM25 (Best Matching 25) algorithm. This algorithm is used in Elasticsearch as well. BM25 is a probabilistic algorithm that ranks documents based on their relevance to a given query. It takes into account both the term frequency and inverse document frequency of each query term BM25 is known to be effective for ranking search results in a wide range of use cases including web search and information retrieval.

In Elasticsearch, BM25 is used as the default scoring algorithm for full-text search queries. However, Elasticsearch also allows you to use other scoring algorithms such as TF-IDF and DFR (Diverted Free Ratio) for more specialized use cases.

Here, is the pseudo-code for the Algorithm.

```
function bm25(query, document) {
```

```
    let score = 0;

    let k1 = 1.2;

    let b = 0.75;

    let avgDocLength = getAverageDocumentLength();

    let documentLength = getDocumentLength(document);

    let documentFrequency = getDocumentFrequency(query, document);

    let termFrequency = getTermFrequency(query, document);

    let inverseDocumentFrequency = getInverseDocumentFrequency(query);

    // This is the formula for calculating BM25 algo

    score = inverseDocumentFrequency * ((k1 + 1) * termFrequency) / (k1 *
((1 - b) + b * (documentLength / avgDocLength)) + termFrequency) *
documentFrequency;

    return score;

}
```

Explanation:

- **query:** The term is being searched.
- **document:** The document is being scored.
- **score:** Final score for that document based on the query.
- **k1:** This is a tuning parameter that controls the impact of term frequency on the score. A greater k1 number indicates that documents with a high term frequency will be given more weight.
- **b:** A tuning value that controls the impact of document length on the score. A higher value of b will give more weight to shorter documents.
- **avgDocLength:** The average length of the docs in corps. That value is precalculated and used as a constant.
- **documentLength:** The length of the current document being scored.
- **documentFreqency:** The number of documents in the corps, that contains the query.

- **termFrequency:** The number of times the query term appears in the current document.
- **inverseDocumentFrequency**: This is a measure of how rare the query term is in the corps. This value is calculated using this formula `log((totalDocuments - documentFrequency + 0.5) / (documentFrequency + 0.5))`, where `totalDocuments` is the total number of documents in the corpus.
- **getAverageDocumentLength():** The function that calculates the average length of documents in the corpus.
- **getDocumentLength(document):** The function that calculates the length of a given document.
- **getDocumentFrequency(query, document):** The function that calculates the number of documents in the corpus that contain the query term.
- **getTermFrequency(query, document):** The function that calculates the number of times the query term appears in the given document.
- **getInverseDocumentFrequency(query):** The function that calculates the inverse document frequency for the given query term.

Elasticsearch query for searching:

searchOpenAPI method that I showed you below, there I discuss Elasticsearch code for searching it takes parameters: keyword and contentType. The keyword parameter searches for OpenAPI documents that have the given keyword in their content field, and the contentType parameter filters the search results based on the defined content type.

The function uses the client.search method to submit a search request to Elasticsearch, sending in the index name and a query object that specifies the search criteria. To match the content field and contentType field, the query object employs the Elasticsearch Query DSL, particularly the true query.

```
async searchOpenAPI(client, keyword) {
       try {
           const response = await client.search({
               index: 'openapi_docs',
               body: {
                   query: {
                       bool: {
                           must: [
```

```
                                { match: { "body.content": keyword } }
                            ]
                        }
                    }
                }
            });
            if (response.hits.total.value === 0) {
                throw new Error()
            }
            return response.hits.hits
        } catch (error) {
            return [];
        }
    }
```

- **Validating Service:**

  When a user submits a search query, the application responds with a collection of OpenAPI definitions. Because the OpenAPI definitions returned by the search engine may have been scraped and indexed previously, it is critical to validate them at the time of answer to ensure that they are current and valid.
  This ensures that the OpenAPI definitions returned by the search engine are not only pertinent to the user's query but also valid and usable. Furthermore, if any of the OpenAPI definitions are found to be invalid during the validation process, they can be removed from the search results, improving the search engine's general quality.
  Validating the OpenAPI definitions entails ensuring that the OpenAPI specification document conforms to the OpenAPI specification's standards and regulations. RESTful APIs are described in OpenAPI definitions, which function as a contract between the API provider and the API user. The OpenAPI specification is intended to encourage API application uniformity, interoperability, and simplicity. When validating OpenAPI definitions, ensure that the specs are correct. This includes ensuring that the API definition includes all of the necessary components, such as API endpoints, request/response parameters, data kinds, and HTTP methods.
  OpenAPI specifications can be validated using tools such as the Swagger Editor and a nodejs module like **swagger-parser which is the key focus of**

our solution. These utilities will examine the API definition for mistakes, warnings, and inconsistencies.
This is a basic validation example, using the **swagger-parser** npm package which is intended for this purpose. It can take an OpenAPI definition URL as an input and validate it against predefined standards using the validate() method.

```
SwaggerParser.validate(OPENAPI_DEFINATION_URL)
 .then(() => {
   console.log('OpenAPI definition is valid!');
      // start indexing the data
 })
 .catch((err) => {
   console.error('Error validating OpenAPI definition:', err);
 });
```

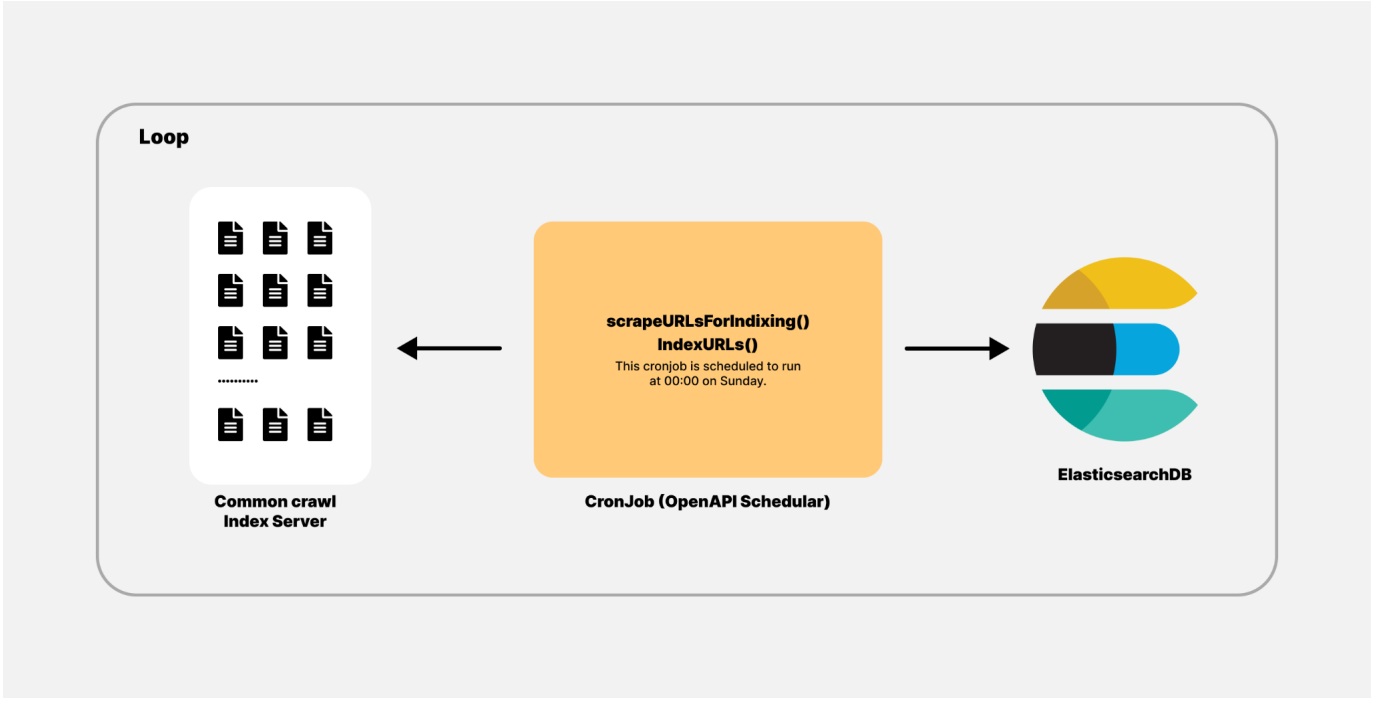
- **Updating dataset Service:**

  It is necessary to update our database with the most recent OpenAPI documents to conduct effective and relevant searches. To do this, we will use the cronjob method. Cron is a time-based job scheduler in Unix-like operating systems. It allows users to schedule recurring tasks, such as running a script or executing a command, at specific intervals. A cron job consists of a cron expression and a command that is executed when the expression matches the current date and time.
  Our program will automatically update our dataset once per week. The Cron Job (OpenAPI Scraper) is responsible for periodically fetching the index files and scraping them for indexing. For doing that we can use the **node-cron** module in nodejs.
  Here is a simple example:

```
startCronJob() {
       getDataFromCommomCrawlServer(URL).then(async res => {
           const ccFilesPromises = res.map(async r => await
getDataFromFiles(r));
           const results = await Promise.all(ccFilesPromises);
           const links = results.flat()
           readContent(links, 0)
       })
```

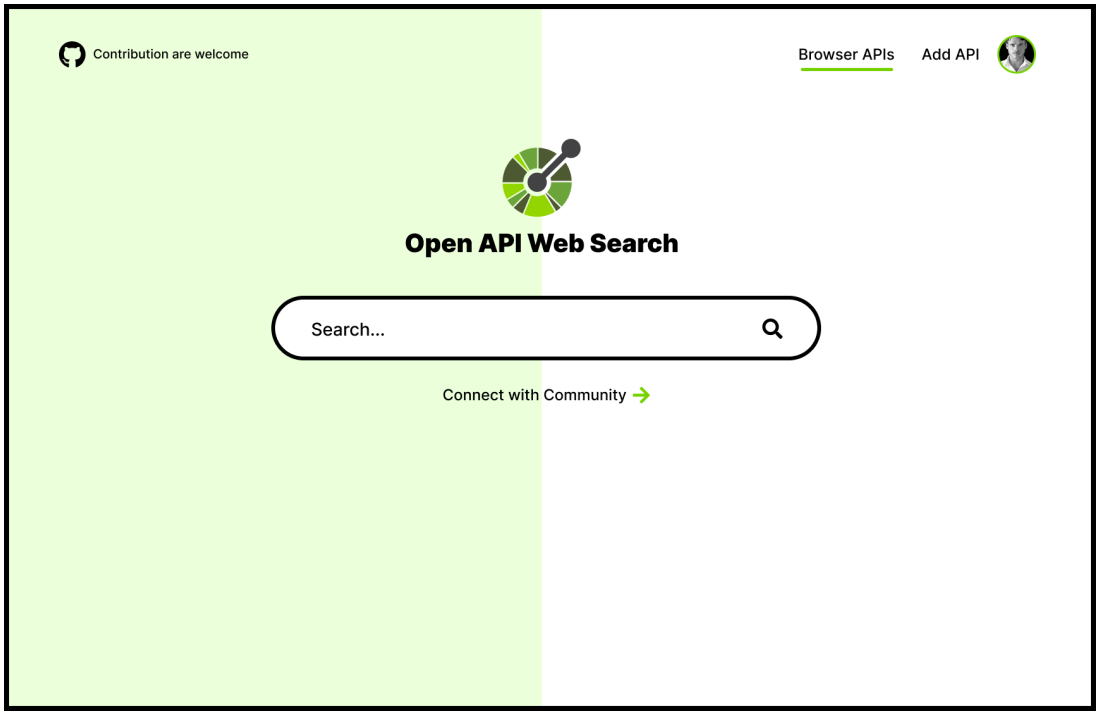
```
        cron.schedule('0 0 * * 0', () => {
            getDataFromCommomCrawlServer(URL).then(async res => {
                const ccFilesPromises = res.map(async r => await
getDataFromFiles(r));
                const results = await Promise.all(ccFilesPromises);
                const links = results.flat()
                readContent(links, 0)
            })

        })
    }
```

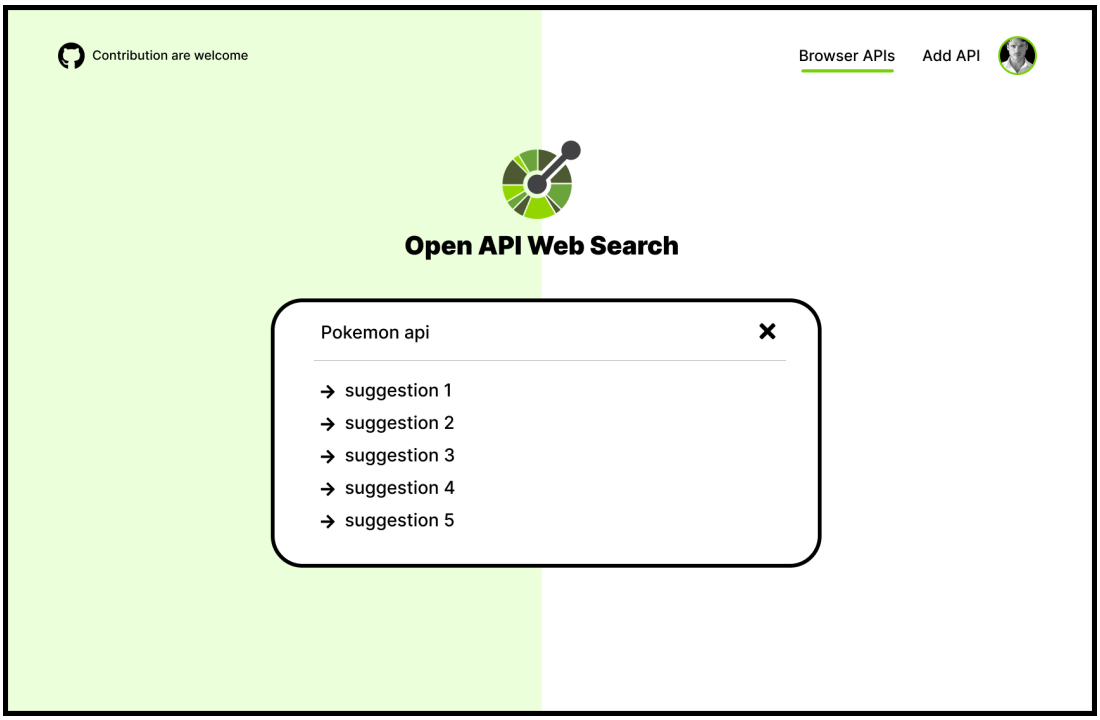
The `startCronJob()` function schedules a recurring task to retrieve data from Common Crawl files and process it using the `readContent()` function. It first retrieves a list of files from a specified URL, then uses `map()` and `Promise.all()` to asynchronously process each file and flatten the resulting array of links. The `readContent()` function is called with the flattened links array and a starting index of 0. A cron job is set up using `cron.schedule()` to run this same process on a regular schedule.



Application UI/UX -

Here are some UI/UX samples for our search engine's Frontend. These illustrations were created using Figma. Source: [Design URL]





Search...

Search Results

t is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout.

openapi swagger

Search Results

t is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout.

openapi swagger

Search Results

t is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout.

openapi swagger

Search Results

t is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout.

openapi swagger

Search Results

t is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout.

openapi swagger

Search Results

t is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout.

openapi swagger

Search Results

t is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout.

openapi swagger

What do I expect from participating in GSoC?

TLDR;

- Community
- Connections
- Guidance

I spent most of my computer science career in a tier 3 city in India, where there isn't much of a coding culture. Most people around me aren't coders, and those who are, are casual coders. I have dreamed to connect with a good coding community, and participating in GSoC could provide me with that opportunity. Without access to a good community, it is clear that I don't have many connections in the field. As a coder, having good connections can significantly boost my career, and GSoC can help me to build new connections. Up until now, I have done everything on my own, with considerable help from the Internet. However, I now aspire to work with real coders and engineers who are masters in their craft, from whom I can learn a great deal. GSoC would offer me the chance to collaborate with these exceptional individuals.

Timeline -

Time Commitment:
In 12 weeks time frames, I would be able to devote ~50 hr/week ~60 hr/week. may increase if the need arises.

Community Bonding:
Gaining a greater understanding of the team's functioning and members' abilities to determine my best place on the team and maximize my contribution, I plan to collaborate with my mentors to gain a better understanding of Postman coding practices and principles. In addition, I would appreciate their guidance on the algorithm and strategies for scaling the system to accommodate a larger user base. Before beginning work on the solution, I believe it is crucial to learn more about the potential user base and gather insights from mentors. Finally, I intend to review and refine our current approach through a brief discussion with both the team and mentors.

- **Week - 1:**
    - Start coding and comprehend the necessary component of the answer. begin planning our 12+ week journey and developing the web application architecture. determine the end product's appearance, the number and type of views that should be created, etc.

- **Week - 2,3:**
    - The first steps in implementing the crawling service involve setting up the development environment and writing a script to crawl index files. The script will be intended to parse the content of these files, to identify target files for indexing. The target files will be indexed after being found to make a search engine.

- **Week - 4,5:**
    - To further develop the solution, the next step is to start writing the APIs. The front-end and back-end components of the solution will be able to communicate thanks to the APIs. Specifically, an endpoint for search will be created, and the search query will be implemented using Elasticsearch. This will allow for efficient and accurate searching of indexed files.

- **Week - 6,7:**
    - After finishing the setup of the index data and search API, the next step is to optimize the code and build a basic test suite for the backend. The optimization process will involve improving the efficiency and performance of the code, while the test suite will be intended to ensure that the backend works correctly and reliably. We can enhance the solution's effectiveness and usability by putting these steps into practice, which will also increase its quality and dependability.

- **Week - 8,9:**
    - Now that the backend is established, we can move on to developing the solution's frontend. This involves designing the user interface and ensuring that it is intuitive and easy to use. After the user interface is created, we will test it to make sure it works properly and offers the

required features. We will have a fully operational and user-friendly solution that satisfies the requirements of our users once we have finished these steps.

- **Week - 10, 11, 12:**
    - Take feedback from the community and iterate on the designs and improvise on use cases. Ensure code quality by adding more test cases and working with more stuff. Work to make documents, blogs, or videos to help increase the user base for this product(Subject to developer community approval).

- **Week - 13:**
    - Spare week in case of some work gets delayed, in case of any emergency or otherwise.

Note: While making the proposal, I have taken the reference from the 2022 GSoC Proposal made by Shivam.

References -

- Common crawl
- Postman API first world
- APIs guru
- Elastic search documentation
- OpenAPI Initiative
- Swagger hub
- Nodejs documentation
- OpenAPI Github
- Elasticsearch youtube
- GSoC Website
- Full-text search Wiki
- Regex Compiler
- Hostinger: Cronjob Comprehensive guide
- crontab.guru
- chatGPT