Cardiff University
Computer Science and Informatics

CM3203 – One Semester Individual Project – 40 Credits

Idiom Search Engine

Final Report

Author: Callum Hughes
Supervisor: Irena Spasic
Moderator: Martin Caminada

Table of Contents

Abstract

Idioms are phrases interoperated in a figurative sense; these can often be modified during use. As idioms can often vary, this makes finding all the occurrences for a particular idiom within a text challenging. The project sought to implement an idiom search engine. The search engine needed to be capable of quickly retrieving documents matching a search phrase with high overall performance for precision and recall. The open-source search engine technology Elasticsearch, and other supporting technologies, were used to implement a solution successfully. In terms of F-measure the solution showed high overall performance when compared to a standard string-matching approach with results calculated as 98% against 39% retrospectively. At the time of writing the application is available to use by visiting the following URL: www.idiom-search-engine.com

Introduction

Idioms are phrases (i.e. groups of words) whose meaning may not be deducible from those of the individual words (e.g. ‘bury the hatchet’ = ‘end a quarrel or conflict and become friendly’). One difficulty in finding idioms in text is the fact that they can vary, so a simple string search is not appropriate. For example, searching for “bury the hatchet” would miss this idiom in the following two sentences:

‘Christmas looks to be a time for *burying the hatchet* or exhuming it for re-examination.’

‘From the look of things, the *hatchet has been long buried*.’

The project aims to create an idiom search engine, which takes an idiom as input and finds all of its occurrences in a corpus of text. The aim is to provide better overall performance than using a string-matching technique seen in existing search tools[1]. By doing so, this can provide linguists with a valuable tool for analysing how a particular idiom might be used within language.

Previous research concerning recognising idioms in a text describes two possible approaches. The first approach[2] details the use of a set of manually defined lexico-semantic pattern matching rules. This hand-crafted collection of patterns for a list of 580 idioms represents the gold standard in terms of recognising a particular idiom in text. However, it comes with the overhead of finely developing and testing each idiom’s particular pattern. The follow-up approach to the previously mentioned study[3], aimed to develop a method for automatically recognising an idiom in text. The method consists of a set of five rules used to create a pattern suitable for any given idiom. The main approach of the project is to use aspects of these rules to support a new method based around an inverted index, allowing high-speed retrieval of results.

Approach

Requirements

The project supervisor also acted as the client for the project. As such, a set of requirements were first agreed upon and are as follows:

Functional Requirements

- The user will be able to input an idiom to search with using a free text field.
- The user will have the option to select an example idiom to search with.
- The system will be able to recognise and retrieve a given idiom within a corpus and display the results to the user.
- The user will be able to download the results as tagged results to process offline.

Non-functional requirements

- Processing of the idiom search will have an efficient response time. (less than 1-minute on average).
- The user interface will have a consistent style and colour scheme.
- The web application will be useable on desktop devices.
- The system should contain no significant security vulnerabilities during OWASP top 10 evaluation.

Once the requirements were established, this allowed work to progress onto planning the possible architecture of the solution.

Corpus

A suitable corpus was first selected to provide a representation of English text when retrieving idioms. The corpus chosen for use in the project was the British National Corpus (BNC). The BNC is a 100-million-word text corpus of samples of written and spoken English from a wide range of sources[4]. The BNC specifically covers British English and allows research on variations between genres, dialects and time periods. As such, this has been the corpus of choice for many studies, including the previously mentioned studies involving idioms.

Elasticsearch

One of the most significant steps towards identifying a viable solution was identifying a technology suitable for high-speed search and retrieval of text. The use of patterns is a key limitation of the method described in the previous studies when applying to a search engine. When searching with a pattern, a string-searching algorithm is used. The Boyer–Moore string-search algorithm has been the standard benchmark for the practical string-search literature[5]. The performance of this algorithm in its most optimal form results in a linear runtime across all cases[6]. Given the size of the corpus, this is likely to be considered too slow to achieve the usability goals set out in the requirements.
Most retrieval engines for full-text search typically use a data structure known as an inverted index[7]. It contains a list of all the terms from a data store in alphabetical order (the index file) and maps them to the documents they appear in (the posting list). Given a user query, the retrieval engine uses the inverted index to score documents that contain the query terms with respect to some ranking model, taking into account features such as term matches, term

proximity and attributes of the terms in the document. This design optimises the query time at the cost of increased processing when a document is added to the database. An example of a technology that manages documents using an inverted index is Elasticsearch. Elasticsearch is a distributed, open-source, full-text search engine with an HTTP web interface and schema-free JSON documents[8]. Because Elasticsearch is open-source and offers a straightforward HTTP interface, a big ecosystem supports it. The core features are:

Distributed and highly scalable: Stored documents can scale to thousands of nodes as data grows. The index is split into shards which are also replicated, increasing availability and providing redundant copies of the data in case of hardware failure.
RESTful API: Elasticsearch works via HTTP requests and JSON data; any language or tools that can handle HTTP can use Elasticsearch. Most languages also have specialised Elasticsearch libraries making development even easier.
Powerful query DSL: Complex queries can be defined easily by utilising the vast features and support provided with the domain-specific query language (DSL).

Elasticsearch was therefore chosen as the database for the project. The ability to store and index all the corpus data and serve complex queries through HTTP, made it the stand-out candidate for the rapid development and full-text search features required for the project.

Additional Processing

Because of the need to potentially build queries using natural language processing (NLP), and avoid the need for novice users to interact with the Elasticsearch API directly, a technology was chosen to provide additional processing of queries. Python is a popular choice for NLP because of the natural language toolkit (NLTK), which is the most popular library for NLP. Python also supports the Elasticsearch client, is easy to distribute through cloud services, and my familiarity with it allows rapid development. As such, I chose Python to provide additional pre and post-processing capabilities.

User Interface

I chose React as the framework/library for developing the front-end of the application. React is a modern JavaScript library that allows component-based development for responsive single-page applications. Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. React is open-source, has detailed documentation and many external component libraries for use in design.
To set the baseline approach to the design of the user-interface, mock-up examples were created and approved by the project supervisor/client (appendix figures 1-4).
As the client-side can support the general functionality of the front-end (except search), a server-rendered build of the website, such as one that uses NodeJS, would not be necessary.

Distribution

For the project to demonstrate the use of web application security practices, hosting and deployment was next considered. Elasticsearch is deployable through any cloud service provider; however, through studying guides[9], it was found deploying, securing and managing scale is a significant overhead over a short period. A managed solution relives this overhead by launching containers with Elasticsearch already installed, also simplifying management tasks like hardware provision, monitoring and network access control. Of the

options available, Amazon Web Services (AWS) provides the Amazon Elasticsearch Service, which is the only managed service available for free; under AWS free tier usage[10]. Amazon Elasticsearch Service offers all the features mentioned above as well as coming with Kibana (a web-based graphical UI that lets you interact the Elasticsearch indices) automatically provisioned. Using the AWS library of solutions, creation of the other components in the system to work with Elasticsearch Service are also straightforward to configure. One service of use is AWS Lambda; Lambda is an event-driven serverless computing platform. It automatically provisions computing resources required by the code and can be triggered by custom HTTP requests configured with AWS API Gateway. Supported languages include Python, which makes it suitable for facilitating the Python pre and post-processing.
The architecture for the project was therefore planned as:

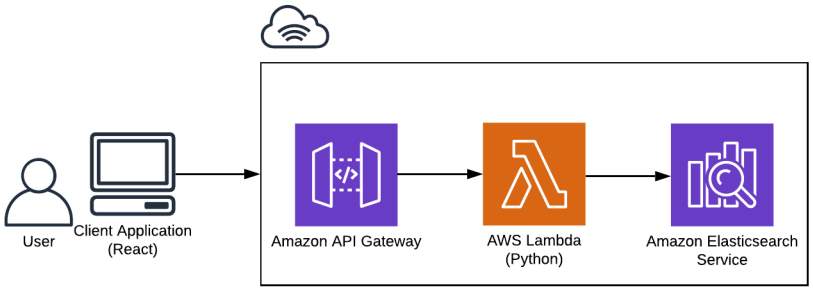

Figure 1: A basic overview of the project multiservice architecture.

A search request made from the React client application is sent through a GET HTTP request configured by API Gateway, which starts the Python Lambda pre-processing. After the input is pre-processed, a search HTTP request is made to the Elasticsearch service where the corpus data is stored and indexed. The results from Elasticsearch are post-processed by the Python Lambda, which finally sends the response back to the client application. To ensure the project achieves the security requirements the principle of least privilege is used when configuring each component of the architecture[11].

Implementation Timeline

The general timeline for the implementation of tasks described in the 'Initial Plan' was kept, given there was no significant change in the perceived architecture of the project. More detail was added to weeks 2, 3 and 4 as it was possible to conceptualise how the processing API would consist of Elasticsearch service and a Python Lambda. Specific tasks from the agreed requirements were mapped to the component of the architecture where it would need implementing, for example, a 'download results page' task could be added to week six where the user interface was planned to be created. A weekly meeting set up with the client was used to discuss and review the progress of tasks and outline the aims for the upcoming week ahead. Notes taken in each meeting allowed progress against the planned timeline to be evaluated at the start and end of each week. All code written over the course of the project was stored in Github repositories to provide version control and back-ups.

Implementation

The first point of implementation was to create an Elasticsearch database and upload documents to test the search functionality. An account with free tier access was used to create an AWS Elasticsearch domain. For free tier usage, a single node and a small tier of instance (3.3 GHz Intel Scalable Processor and 2GiB of memory) can be configured. The access policy to the provisioned endpoints was initially set to whitelist my IP address, with the intention to later refine access to better adhere to the principle of least privilege. Once the endpoints were accessible, I implemented a local Python project to bulk upload data to an index using the Python Elasticsearch library. The data initially uploaded was from a list of over 2000 sentences containing idioms compiled during the previous study[12]. A file containing the correct sentence results for a set of 427 idiom searches (created to test the previous studies) was obtained and adapted to JSON format. Using this file, a test suite was designed using the pytest library to explore the effect a change in query or indexing technique would have on the precision and recall of results. A parameterised test was used to automatically take an idiom from the file, compile a query, search using the Elasticsearch endpoint and compare the results to the expected results in the file, repeating for all 427 idioms. Once all tests completed a report was output for each idiom consisting of: matched results, incorrect matches and missing sentences, a figure for individual test precision and recall, and overall precision and recall. The detailed report allowed the effects and limitations of changes to be better understood. To approach improving the performance of searches, the previous study's five methods were adapted to work within Elasticsearch.

Analyser

A custom analyser can be configured to provide low-level control over how terms in the index are tokenised during index time, the effects of the analyser also run at search time[13]. The custom analyser used for the project first normalises many terms in the index by filtering all characters to lowercase and removing most punctuation, including apostrophes. Each term is then processed to its root word using the Porter stemming algorithm[14], meaning each root word is stored in the index rather than the actual word that appears in the sentence. The query terms also go through the same stemming process when searched, allowing a given search phrase involving inflexion to be matched using the index. This produces equivalent results to the inflexion stage in the previous study, where lemmatisation is used on the canonical form of the idiom and the text to match inflected terms within idiom phrases.

Synonym filters map specific terms or whole phrases to the same index. Idioms often contain slots for the use of pronouns eg. ‘darken his/her/their/one’s door’. Therefore, pronouns in each document are indexed as synonyms. If a query contains a pronoun, this will be searched as equivalent to any other pronoun.

Other synonyms used in the index are synonyms for equivalent idiom phrases. The individual words used in each phrase are often not direct synonyms of one another, although figuratively, they are equivalent in the context the whole phrase and their use within a sentence. For example:

‘bored to tears’, ‘bored to distraction’, ‘bored to death’, ‘bored silly’

These were easily identifiable from the set of idioms acquired from www.learn-english-today.com as they would be listed as a single idiom containing one or a number of slashes eg. ‘beat/flog a dead horse’. Altogether a set of 71 different groupings of equivalences were identified from 2946 individual idioms. Each group of equivalent idiom phrases are mapped as synonyms within the index. This is designed to allow a user to avoid having to search separately for multiple equivalent phrases. As these synonyms are derived from a list, this feature does not conform with the automated approach used elsewhere for identifying idioms and is outside the scope of the rules detailed in the previous study. Due to their potential to improve search for at least some cases, they were kept in the final solution.

A JSON body used to specify the custom analyser, is sent to the Elasticsearch ‘create index’ API before uploading documents. The techniques used in the analyser not only improve the search quality but also reduce the index size by removing the differences between similar words.

Query

When storing sentences in Elasticsearch, an inverted index is created. The position of the term within the document is also mapped (word-level inverted index). This allows exact phrases to be searched, as the index shows not only if the query terms appear in the document but also if they exist in the same order as they do within the search query. A phrase can be queried using a ‘match_phrase’ leaf query clause, sent in a JSON body to the search API[15]. Within the project, a query is processed in a number of ways to create a compound query clause, containing up to 4 separate Elasticsearch Boolean leaf query clauses[16]. Each Boolean leaf query is a ‘should’ clause, meaning it's possible for matching results to exist from a clause or not at all. Each clause of the query can be described as follows:

Idioms are often modifiable; for example, adjectives can be added into the middle of the phrase to modify a noun:

‘snatch victory from the jaws of defeat’
‘snatch *a lucky* victory from the jaws of *almost certain* defeat’

Word slop is a ‘match_phrase’ query option that allows a set number of word positions to exist between matching tokens in search results. The project makes use of word slop to allow for modification. The amount of word slop for a given idiom is based on the number of nouns and verbs that exist in the search phrase; given modifications can precede or proceed these words. Here, Python's NLTK part-of-speech tagging (POS) functions are used to identify this number from the input idiom. The calculated word slop and the standard string of the idiom make up the first clause of the query. This section also represents the only compulsory section for all searches. This stage is designed to be equivalent to the modification stage in the previous study, where POS tagging is used to identify potentially modifiable sections in the canonical form of the idiom.

Idioms often contain open slots where noun phrases can be inserted in the place of pronouns:

‘jog someone’s memory’
‘jog *Cliff’s* memory’
‘jog *the accountant’s* memory’

POS tagging is used to identify pronouns in the search idiom. To provide an open slot, Python is used to replace any pronouns in the idiom with a wildcard. A wildcard is interpreted at query level as an empty space, effectively removing the pronoun and therefore allowing potential noun phrases to be matched in its place. The word slop value previously calculated is increased to accommodate noun phrases that might exist within the space created. The increased word slop and the idiom without pronouns make up the second clause of the query. This stage is designed to be equivalent to the open slots stage in the previous study, where POS tagging is again used to identify where slots exist in the canonical form of the idiom.

Verbs in many idioms may appear in the passive voice:

‘pick someone’s brain’
‘someone had offered their brain to be picked’

To automatically generate a passivised version of an idiom, POS tagging is used to identify non-auxiliary verbs at the beginning of an idiom. The verb is repositioned to the end of the idiom, and a space inserted in front of it. The amount of word slop is increased for the new query format to accommodate the space introduced. The passivised form of the idiom and the increased word slop makes up the third clause of the query. If the idiom also contains a pronoun, this suggests a passivised version of the idiom with a noun phrase inserted in the place of the pronoun is possible. Therefore, a fourth separate clause can be used containing the idiom without pronouns and a further increased slop.

An example of an idiom where all four clauses are applicable is shown in appendix figure 5. If a certain clause does not apply to the search idiom, then the cause is not added to the final compound query for that search.

Order of Results

The order matching documents appear in the search results is based on a relevance score, the larger the score, the higher up the list the result appears. A similarity model defines how matching documents are scored. The similarity model used in the implementation is the default Okapi BM25 model; there is no artificial boosting applied to different clauses of the query. Conveniently, documents matching the first clause of the query will score highest (appear first). This is because sections involving open slots or modifications (all other clauses) concede at least one word provisioned by slop will be required to match documents.

Uploading the BNC corpus

With the method of identifying an idiom in text tested, the BNC was uploaded to Elasticsearch. The corpus was sourced from the oxford text archive in XML format. The project previously used to bulk upload the sentences list was adapted to read in sentences from the BNC XML using the NLTK BNC reader package. Once read, a sentence was detokenised using the NLTK TreebankWordDetokenizer package, creating a standard readable form before being uploaded with the Elasticsearch client. With the full BNC corpus uploaded this provided 6,026,276 unique corpus sentences available for search.

Python Lambda

Methods created within the test suite were transferred for use in an event-driven Python Lamba function. The Lambda was tested locally then deployed using the AWS console UI.

To continue the principle of least privilege for components, the IAM role for the function was added to the access policy of the Elasticsearch domain only authorising GET requests:

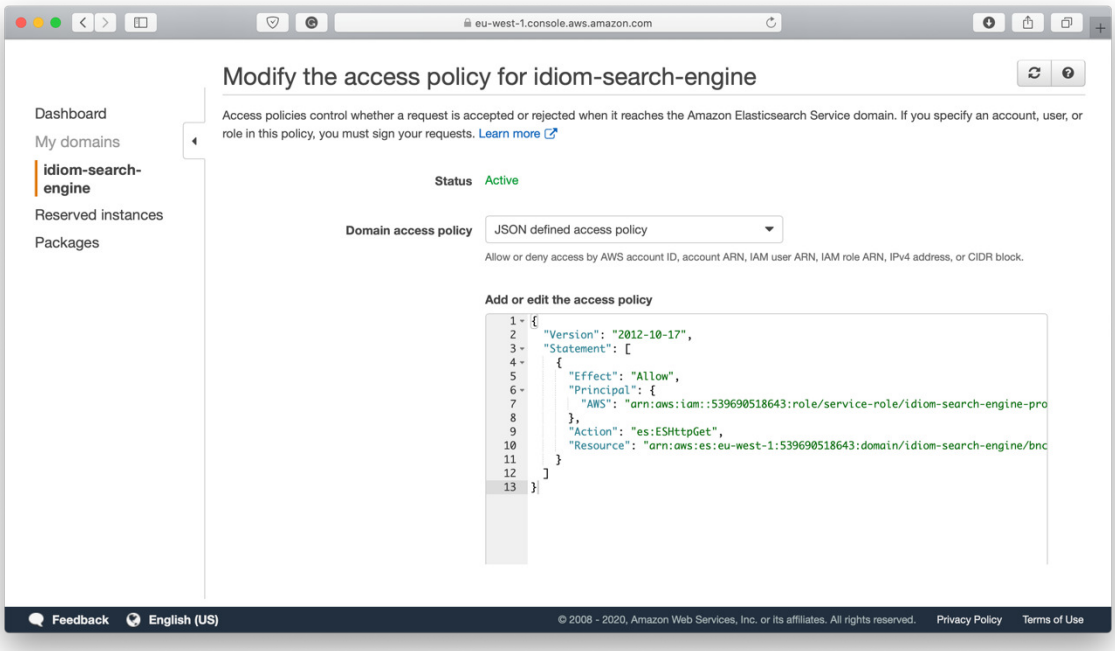

Figure 2: A screenshot of the access policy for the Elasticsearch Service, updated to only allow GET requests to the IAM role associated with the Python Lambda project.

A basic test was provisioned to run the function once deployed. With deployment of the lambda function complete, an API gateway GET trigger was configured to handle the request and response and provision an HTTP endpoint:

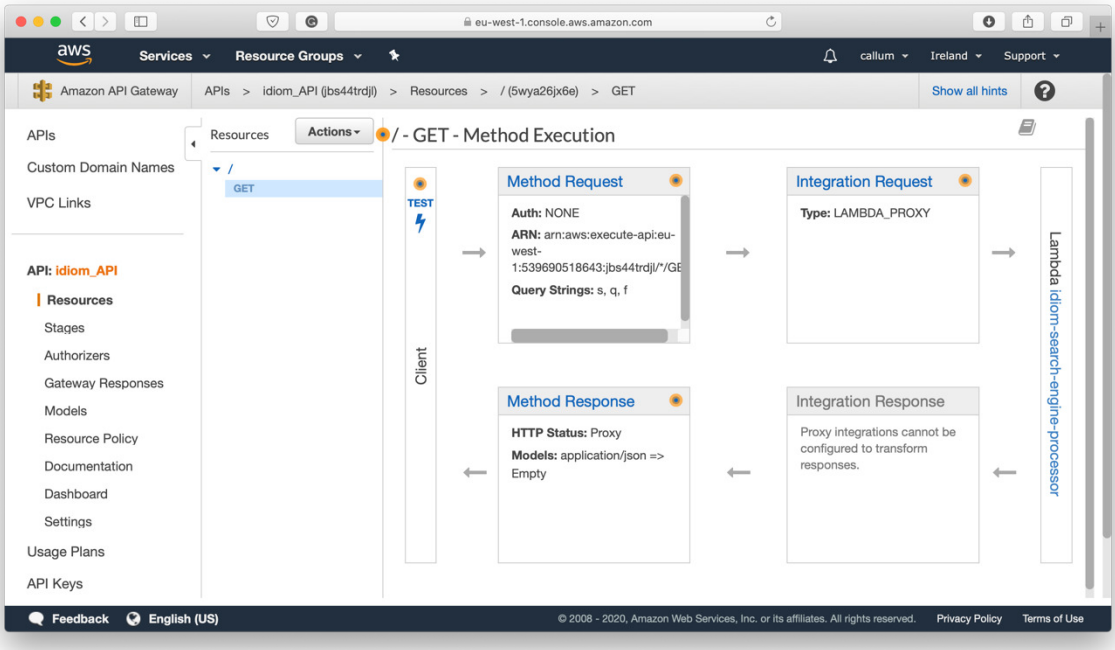

Figure 3: A screenshot of the API gateway GET method configuration, displayed within the AWS management console.

Accepted parameter names and types for the request were configured to save the overhead of managing unexpected parameters manually.

The requirements outline the need for highlighted idiom results in an XML style tag. As such, the Elasticsearch query was modified to provide results with custom highlight tags using the built-in highlighting feature. One issue with the highlighting feature is it tags each word token in a separate set of tags, whereas the project requires a single set of tags covering the whole phrase[17]. Post-processing functions were added to the Python Lambda to provide a solution; using a regex pattern, the text between the first and last 'idiom' tag in a result could be selected. Basic string processing could then remove any other 'idiom' tags inside the matching section. An unknown issue when testing the highlighted results was that individual word tokens in a 'match_phrase' search with slop, could be matched multiple times in a document, provided they existed in the word slop range. This is especially erroneous when a pronoun exists in the idiom, meaning synonyms for pronouns within slop range of search idiom also get highlighted. For example, a search for 'the apple of your eye' would provide the following result:

```
<idiom>He</idiom> was <idiom>the</idiom> <idiom>apple</idiom>
<idiom>of</idiom> <idiom>her</idiom> <idiom>eye</idiom>, <idiom>the</idiom>
salt of her earth, the source of her strength — her everything.
```

Here the processor calculates the word slop as 3 (2 nouns + 1) and given the contents of the search phrase, the beginning pronoun 'He' and trailing 'the', exist within the word slop range and are therefore also tagged. With the basic post-processing, this would leave everything between the first and last tag as the highlighted phrase:

```
<idiom>He was the apple of her eye, the</idiom> salt of her earth, the
source of her strength — her everything.
```

To solve this issue, I improved the post-processing regex pattern to select the opening 'idiom' tag followed by the first word in the query, and the closing 'idiom' tag proceeding the last word in the query. For example, the pattern generated to identify the correct highlight tags for the previous search would be (document highlights show parameters used to generate each pattern):

```
(?i)^(.*)(<idiom>the.*(.*\/idiom>){4,}.*eye.*?<\/idiom>)(.*)$
```

The use of a quantifier allows the pattern to work as expected even when the first or last term of a query appears multiple times in the results. To allow for potential modification of the first and last words to match the pattern, I used the NLTK Porter Stemming algorithm to provide a stemmed version of the first and last word in a query. This type of pattern generation tends to fail when the first or last word in the query is a pronoun, meaning the start or end word cannot be singled out with a pattern easily. It also cannot be applied to a passivised occurrence of the search idiom. If this is the case, the results are returned with the basic tag processing, which can mean some errors in highlighting still exist.

User Interface

Development of the front-end application began with 'create-react-app', this command automatically provisions a development environment with the latest Javascript features for creating a React single-page-application (SPA). Using the mock-up design and requirements,

the necessary routes for the application could be determined. The ‘react-router-dom’ handles switching top-level components based on the URL. The routes defined are:

- exact path "/" switches to the home page top-level component
- "/about" switches to the about page top-level component
- "/results.txt" switches to the download results top-level component
- all other routes switch to a page not found top-level component

The homepage and about page are wrapped in a standardised layout consisting of the header component (which includes the logo and title) and the main body of the component set to a particular max-width. This provides a consistent design with both of the main UI pages. To further implement a uniform design, base components were selected from the component library Evergreen[18]. Evergreen is an open-source React UI framework which contains a comprehensive list of customisable dynamic components and styles for use in the design of a website. As previously described, components allow the UI to be split into interdependent, reusable pieces. The components written in the project conform to the latest standard of approach, which is function components where the component state is handled using React Hooks[19]. Function components are named so because they fundamentally represent JavaScript functions rather than classes; they accept a single ‘props’ (which stands for properties) object argument with data and return a React element. React Hooks allows a function component to use state and other react features through an internal API. With React components, data flows down from parent to child component. A component may choose to pass its state and functions down as props to its child components. When a parent component updates through an event, React detects the change in the DOM and re-renders the component and any relevant child components, allowing flowing, responsive web pages. Figure 4 aims to map out the main components of the front-end application:

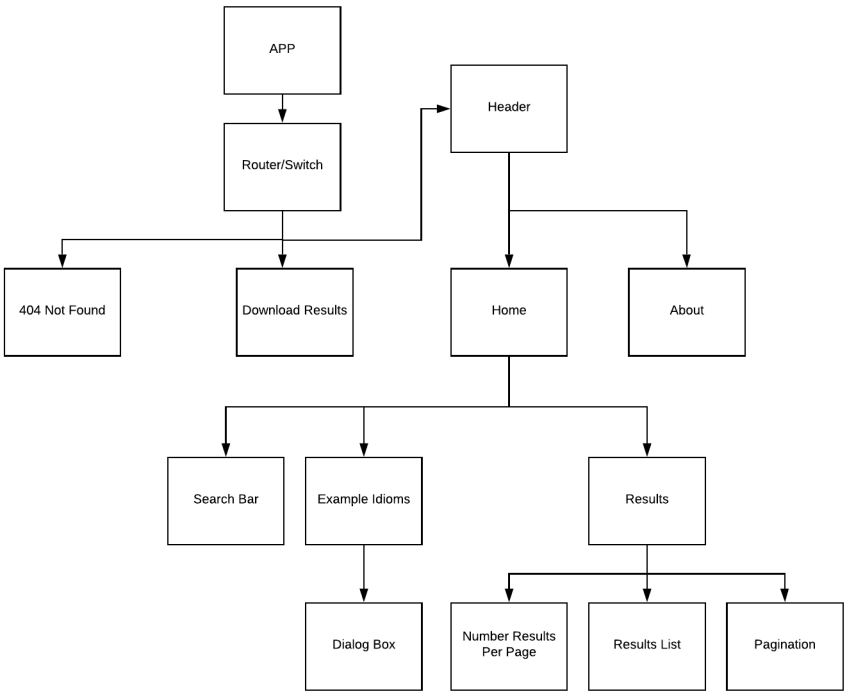

Figure 4: A high level overview of the main component structure of the user interface.

The home page handles the main flow of interaction with the project; as this is where users can search for a given idiom. The search bar component has access to a function for updating the query search state variable, received via props of the home page top-level component. The submission of the search bar form triggers a function to make an HTTP GET request to the API Gateway endpoint using the 'axios' library. A loading spinner renders whilst the request is waiting. Each result in the response is mapped to display the results list (figure 5).

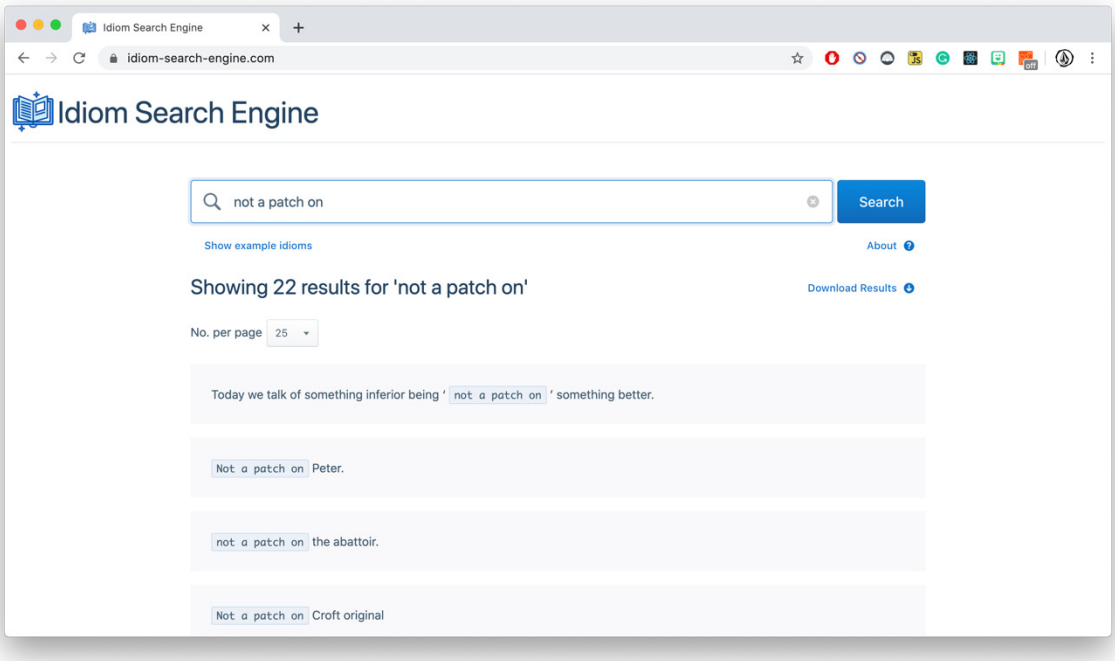

Figure 5: Screenshot of highlighted results shown below the search bar for a given search phrase.

Once the results are displayed, options to change the number of results per page, pagination options, and a link to download the results become available to the user. Changing the number of results per page will update the state and run functions to regenerate the pagination options and retrieve the results again. This state will persist between different searches in the same session. The user can interact with the pagination buttons to retrieve the next set of results; a use effect hook monitors interaction with these elements and re-renders results as they change. The pagination options also regenerate based on a function that maps page number options evenly either side of where the user is currently positioned (when positioned greater than 6). The 'Download Results' button links to a new tab routed to '/results.txt'. URL parameters pass on the state of the search query to run a request for all results (maximum 10000). The results are mapped to the page in raw form in a plain HTML format to read as a basic text output (figure 6).

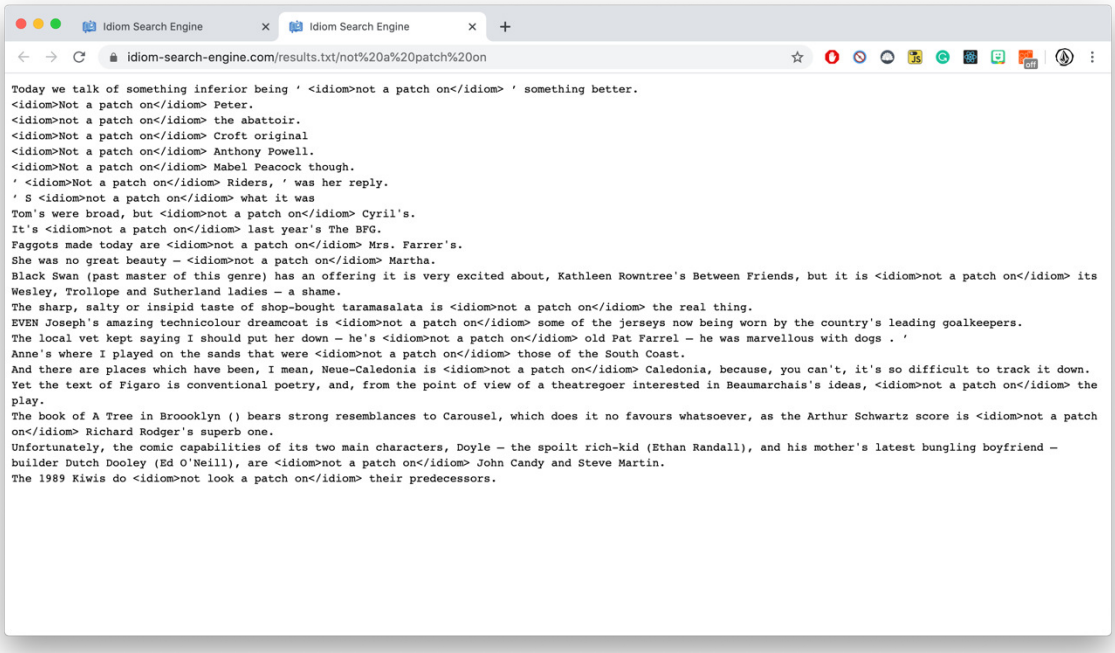

Figure 6: A screenshot of the 'Download Results' page output for the previously searched idiom phrase (shown in figure 5).

The user also has the option to search using an example idiom (figure 7):

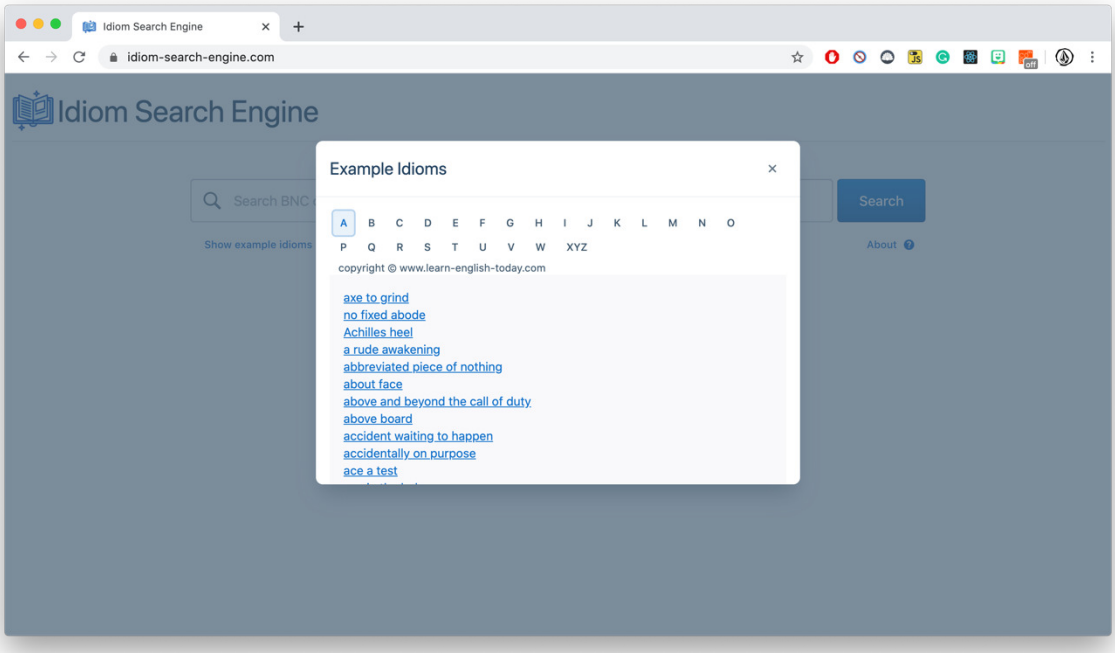

Figure 7: A screenshot of the 'Example Idioms' dialog box displayed once a user clicks the 'Show example idioms' button on the home page.

To gather example idioms, I made contact with the website www.learn-english-today.com to ask permission to use their collection of idioms on the project website. To retrieve the list of idioms in a suitable format, a web crawling project was created using the Scrapy framework. Scrapy is an open-source web scraping framework for Python. It manages requests, parses

HTML webpages, collects data, and saves it to the desired format. This was useful as the list of the idioms on learn-english-today was separated into many individual pages. The written Scrapy spider first requests a page from learn-english-today that lists each link for each separate page the idioms exist. The list of links is used to make a new Scrapy request for each page. Each new Scrapy request runs a function which selects the HTML elements of value on the page using an XPATH. Scrapy then outputs the results to a JSON file. The JSON output file was transferred to the React user interface project to be used locally to render the example idiom results. Here an Evergreen dialogue box component is customised to display the list of idioms separated by alphabetical tabs. The list of idioms can be scrolled through and once an idiom is selected a set of functions close the dialogue box, fill the search bar with the chosen idiom, and update the focus to the search bar component.

The ‘About’ button routes to the ‘/about’ page. This page describes the aim of the project and has a less detailed explanation of how search works. This page aims to help non-technical users understand how their queries are interpreted and how to get the best use from the project (figure 8).

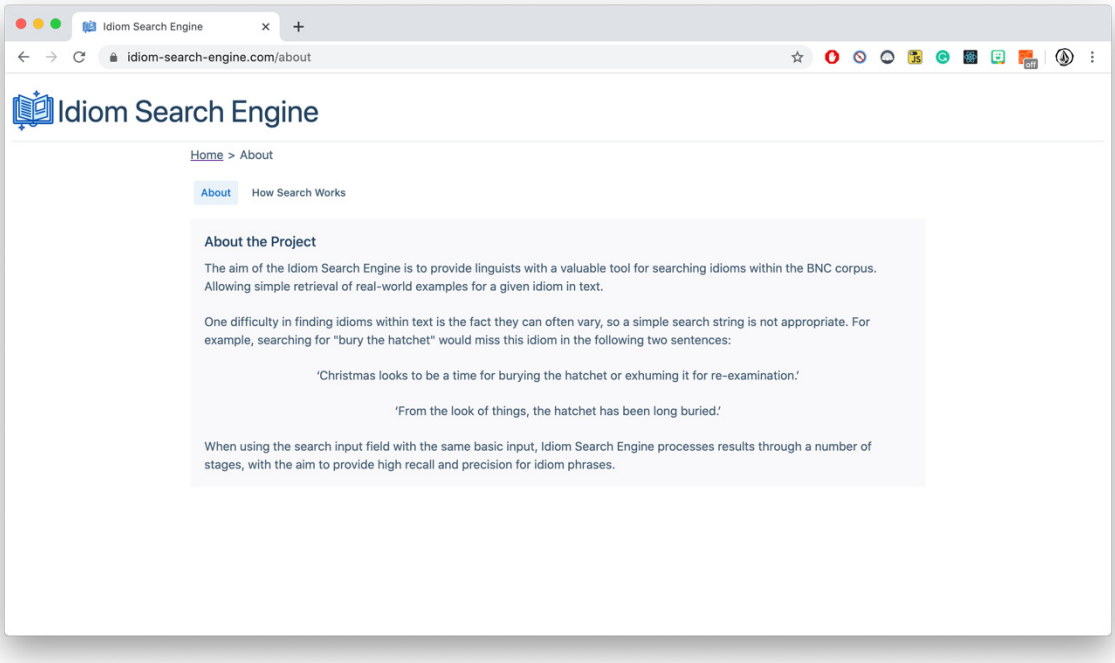

Figure 8: A screenshot of the ‘About’ page.

Deployment

To create a production build of the react web application, the ‘npm run build’ command was used to create a build directory containing the base files of the project. The build directory was saved in an AWS S3 bucket. The S3 bucket was configured to serve the build files through static web hosting. Static websites are very low cost, provide high-levels of reliability, require almost no IT administration, and scale to handle enterprise-level traffic with no additional work[20]. To handle routing, an AWS Route 53 hosted zone was created with the registered domain ‘idiom-search-engine.com’. AWS CloudFront was used to improve the performance of the distribution as well as handle routing users with SSL/TLS certificates. CloudFront makes the website files available from data centres around the world (called *edge locations*)[20]. The static hosting architecture is shown in figure 9.

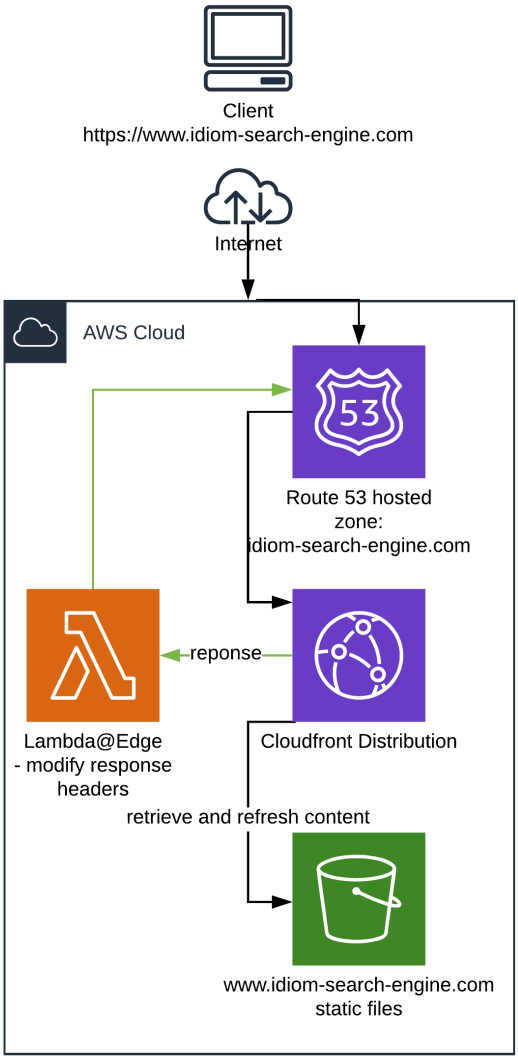

Figure 9: Overview of AWS cloud services used for handling distribution of the static web page files.

Cloudfront can also be configured to run small scripts of code used to alter the response headers for the served website resources through Lambda. This was used to configure Content Security Policy (CSP) headers. CSP is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks[21].

Results and Evaluation

Performance of the Idiom Search

A commonly used method for measuring the performance of NLP technologies is precision and recall, where precision is the percentage of relevant documents amongst the retrieved results, and recall is the percentage of the total amount of relevant documents actually retrieved, calculated as:

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|}$$

$$recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documetns\}|}$$

As mentioned previously, a Python test-suite was written to compare the project's approach against the gold standard approach. Tests contained a search idiom and the expected results for searches made against the 2530 idiom phrases compiled in the previous study. Results showed 15 out of 428 idiom searches did not achieve 100% precision and recall against the test cases. The total figures for precision and recall across all tests came to 96.29% precision and 98.71% recall. Upon closer inspection of the results, incorrectly retrieved documents resulting in lower precision tended to be the result of the flexible query matching rules used in conjunction with short idiom phrases containing common verbs, and pronouns. This type of occurrence is considered the payoff of having a model aimed more towards high recall, as making less flexible matching rules was shown to have a higher detrimental cost to the recall value. Many unmatched documents resulting in lower recall were found to use synonyms in place of nouns and verbs. Noun and verb synonym support for the Elasticsearch index was experimented with during the development of the project. The full list of WordNet synonyms (212000 words) was added to the analyser, to group with uploaded documents. Adding this many synonyms took the single node cluster offline when trying to upload documents. I tested the same implementation using a locally provisioned Elasticsearch cluster with greater hardware resources. The results with synonym support showed extremely low precision, and it was difficult to derive the search query amongst the highlighted output. One way of reducing the size of WordNet synonyms is using word embeddings; however, the project supervisor advised this would be out of the scope project. It was concluded comprehensive synonym support would not be included in the solution, given the detrimental impact on precision and database performance. The overall performance against the tests showed a sufficient balance of precision and recall, however, it was acknowledged by the project supervisor that the test cases used were not extensive enough to provide an accurate representation of performance over a full corpus.

One practical method for testing the results against a full corpus would be to use relative recall against another solution such as the gold standard, and the precision measured by manually inspecting the retrieved results. As this required manually inspecting possibly thousands of results, this type of evaluation was also deemed out of the scope of the project. Therefore, an approach comparing the project search to a standard string search (comparable to existing platforms) was designed. A set of ten different idioms were chosen for comparison. Frozen idiom phrases (idioms that appear in a fixed form) were not chosen for the test, as the aim is to explore the level of improvement for idiom phrases where modification is possible. Each idiom in the list was searched over the BNC using the project search and the string search. The results were collected and annotated. For each idiom, the precision and relative recall is measured to compare each search method (table 1).

Given the string search recall existed as a subset of the project search recall, the relative recall comparison of the project search is always 1. The precision of the string search during annotation was also always 1.

TABLE 1: Project Search Comparison to String Search for Sample Inputs

Idiom	String Search – Precision	Project Search - Precision	String Search - Relative Recall	Project Search - Relative Recall
At someone's beck and call	1	1	(0/42) 0	1
Best of my ability	1	1	(27/93) 0.29	1
Refresh someone's memory	1	1	(0/52) 0	1
The rest is history	1	1	(21/37) 0.57	1
The world is your oyster	1	1	(2/12) 0.17	1
Take for a ride	1	1	(0/29) 0	1
Pick someone's brain	1	(24/27) 0.89	(0/24) 0	1
Turn over a new leaf	1	1	(5/14) 0.36	1
Writing on the wall	1	(54/61) 0.89	(29/54) 0.54	1
Lift a finger	1	(35/40) 0.88	(15/35) 0.42	1
Total AVG:	1	0.97	0.24	1

The overall performance for precision and recall can be expressed using F-measure:

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

In terms of F-measure, the performance of the project search in comparison to string search improves from 39% to 98%. This demonstrates that providing the option of an idiom optimised search engine for idiom related searches offers higher performance than a standard string matching approach.

Performance of User Interface

To measure the performance of the user interface a questionnaire was used based on the System Usability Scale (SUS), a questionnaire for assessing the perceived usability of interactive systems [22]. A set of 5 students were asked to carry out a list of tasks representing the typical user story (search for an idiom and download the results) and answer the questionnaire. It consists of 10 questions based on a 5-point Likert scale (1=strongly disagree, 5=strongly agree).

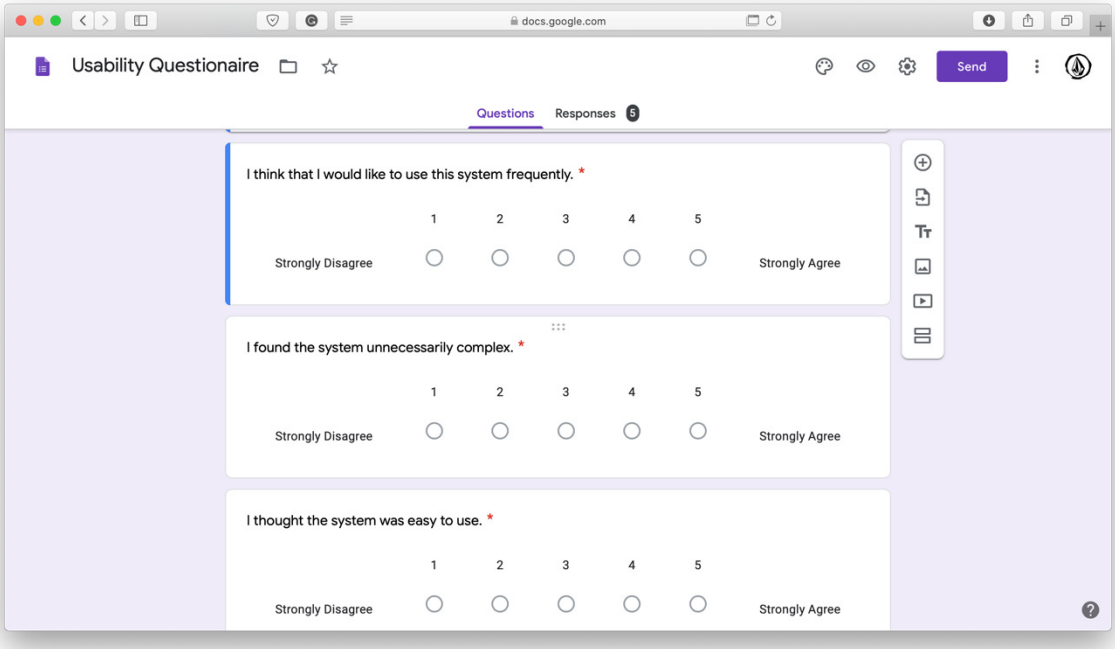

Figure 10: A screenshot of the usability questionnaire sent via Google Forms.

This questionnaire is shown to be reliable and straightforward in determining website usability[22]. The overall SUS score is calculated on a scale from 0 to 100. The score for the project calculated from the student's responses was 96. This score, in comparison to other systems using the same measure, puts the system in the percentile range of 95-100% for usability. This rank can be interpreted as 'Grade A' on a scale from A to F [23].

Web Application Security

As outlined in the project requirements, the web application should contain no significant security vulnerabilities during OWASP top 10 evaluation. The OWASP top 10 identifies some of the most critical risks facing organisations. To evaluate potential OWASP top 10 vulnerabilities existing in the application, a manual inspection and review has been created. This details the possible vulnerabilities and the preventative measures made during the

development of the application. The advantage of using manual review is its completeness and effectiveness, as the source code and architecture is well known by myself. It's also fast to implement in comparison to an automated test. Table 2 lists the most recent version of OWASP top 10 (2017) vulnerabilities[24]:

TABLE 2: OWASP Top 10 Evaluation

Security Risk	Vulnerability Assessment	Preventative Measures
1. Injection	The system only stores data which is openly searchable by design, so accessing data without proper authorisation is not a vulnerability. Elasticsearch API's, if left open are still vulnerable to executing unintended commands.	Dynamic queries are processed using strict parameters only ever interpreted as a strings, as configured in API gateway. The structure of the dynamic queries cannot be directly concatenated with a command that contains a different structure. Queries facilitated by the Python Lambda API are only ever authorised to be GET requests. The Elasticsearch API can be directly accessed by no other means than the Python Lambda, as configured with IAM roles access control.
2. Broken Authentication	Authentication is not used in the project and therefore cannot be exploited.	
3. Sensitive Data Exposure	No sensitive data is stored by design.	
4. XML External Entities	The application does not accept XML uploads, and there is no use of XML processors in the application.	
5. Broken Access Control	User accounts are not used in the project, meaning no risk of vulnerability.	

6. Security Misconfiguration	Serverless reduces the need to patch the environment, since the cloud provider controls the infrastructure. The application is still vulnerable to Denial of Wallet and Denial of Service attacks[25].	Calls to the Python Lambda API have a timeout of 30 seconds to reduce risk of Denial of Service attacks. The function's concurrency is also configured to 0 meaning the function always throttles requests, this reduces the risk of Denial of Wallet attacks.
7. Cross-Site Scripting (XSS)	Although user input is used as part of HTML output (in the 'showing results for…' title), React escapes this automatically. Therefore, the application is not vulnerable to 'Reflected XSS'. The application doesn't store user input so 'Stored XSS' is also not a vulnerability. The system is less likely to be directly impacted by 'DOM XSS' given there's no authorisation or sensitive stored data to exploit, however the application could be used as an attack vector to obtain a user's browser local storage.	Enabling a Content Security Policy (CSP) is described as 'effective if no other (XSS) vulnerabilities exist'[24]. Mentioned previously, CSP headers were configured by Edge Lambda when serving the static web files via the CloudFront Edge servers. This creates a default that blocks unknown content but allows exceptions for the application's functions.
8. Insecure Deserialisation	Python, a dynamic language and JSON, a serialised data type, are used in the project. However, there is no exchange of serialised objects from untrusted sources. Data sent to the Python Lambda API uses query string parameters (a primitive data type) not a JSON object, therefore the application architecture keeps this application safe from this type of vulnerability.	
9. Using Components with Known Vulnerabilities	Measures or tests are not in place to automatically scan for vulnerabilities and	Packages were checked for unused dependencies and uninstalled if not necessary.

	update underlying problems, therefore the system is vulnerable to this security risk.	'npm audit' was used to check for known vulnerabilities. Fixes were then applied to update each package. A permanent solution to this issue should be integrated into a continuous integration and delivery pipeline (CI/CD).
10. Insufficient Logging & Monitoring	A vast amount of logging and metrics for the Elasticsearch service and Python Lambda are provisioned automatically by AWS. However, the application is unable to alert for active attacks in real time, therefore the system is vulnerable to this security risk.	Effective monitoring and alerting of suspicious activity could be supported with a commercial web application firewall. This however may not be necessary given the applications overall low business impact for vulnerabilities.

To ensure the security headers added to the response by Lambda Edge processes have worked correctly, the Mozilla Observatory test was used (figures 12 and 13). Mozilla Observatory is a project designed to help developers configure their sites safely and securely. It scores tests based on how well implemented the latest security features are on the site, as recommended in Mozilla's web security guidelines[26].

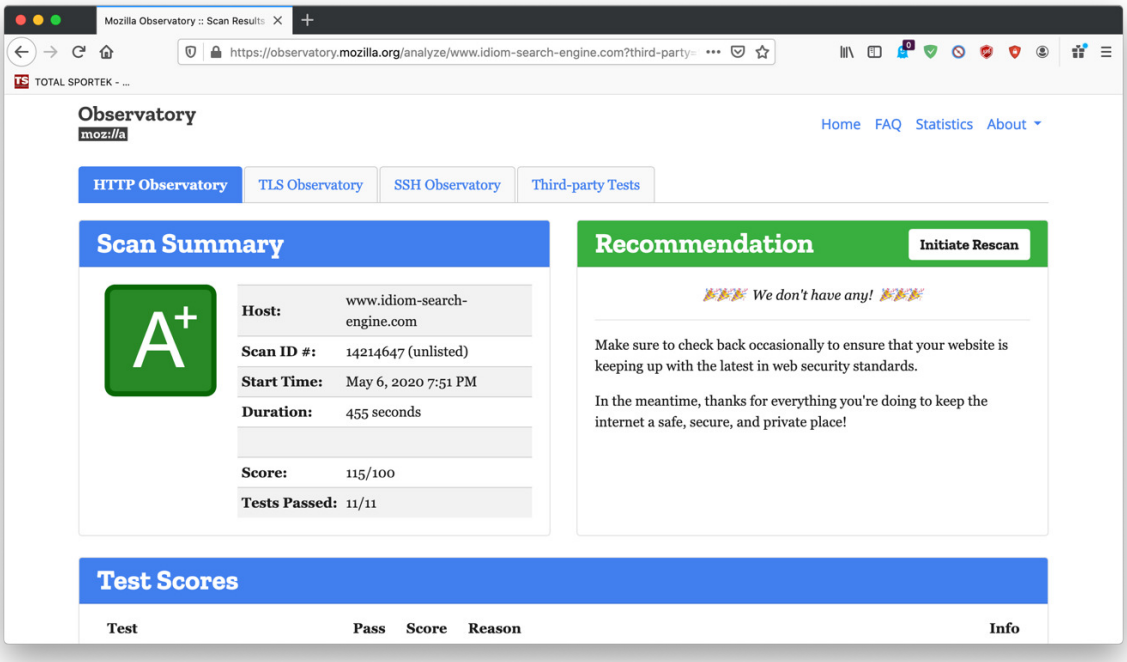

Figure 11: A screenshot of the results summary for the Mozilla Observatory test.

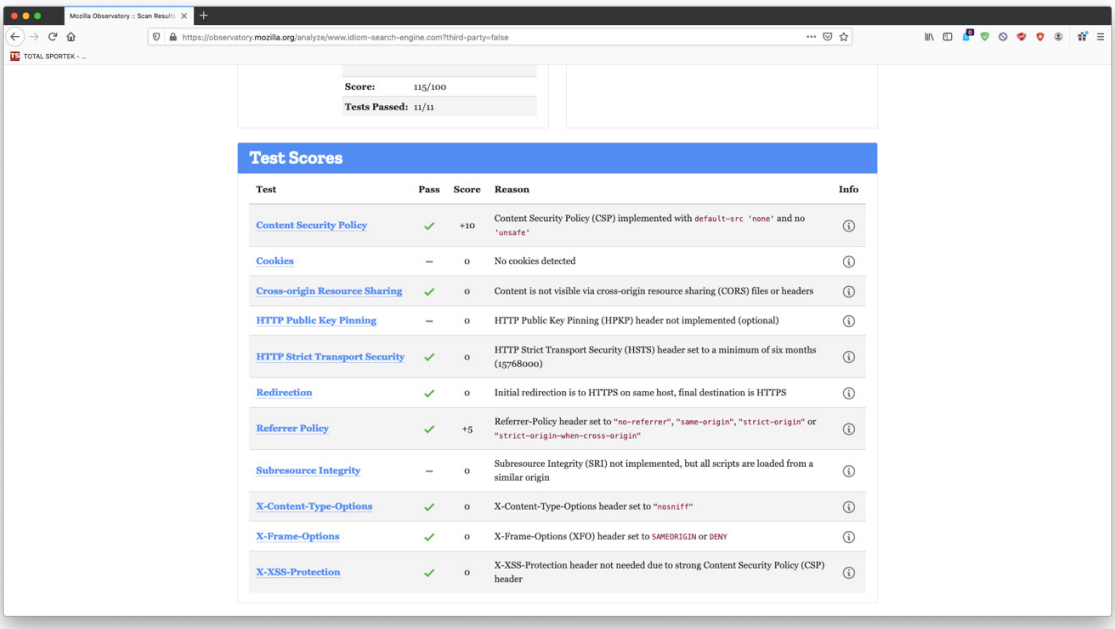

Figure 12: A screenshot showing the results breakdown for the Mozilla Observatory test.

The website scores the highest possible grade for the test showing the latest web security header guidelines are active on the live version of the site.

Given there are at least some preventative measures made against all assessed vulnerabilities apart from logging alerts, I would argue, at least for the time being, that the application contains no significant OWASP top 10 security vulnerabilities. A big advantage towards achieving this requirement was considering security throughout the software development life cycle. This prevented the need to extensively re-model my approach during security

evaluation. However, I acknowledge security implementation was made easier, given the application, by design, doesn't store sensitive data or deal with user authentication. This limits the possible attack vectors and reduces the technical/business risk of vulnerabilities. I also acknowledge that there are many more security vulnerabilities outside of the OWASP top 10, it's almost certain the application is still vulnerable to other attacks. Security also constantly changes; without changing any code, the application may still become vulnerable to newly discovered flaws and attack methods. Updates to security should therefore be made throughout the life span of the application.

Conclusions and Future Work

The project has demonstrated that the methods used for automatically generating patterns can be used to support an approach based around an inverted index. The project search is also shown to have higher performance than existing online approaches to searching for idioms over a corpus. More extensive tests that require collaborative annotation will need to be made to draw further conclusions about the performance in comparison to other approaches such as the gold-standard. The web application was successfully distributed at the time of writing, and is available to use from the following URL: www.idiom-search-engine.com

As the project shows promise in its overall performance for search, the potential to add more features could be explored. One feature of use could be to provide insight into where a retrieved document originates from in the corpus. This could be similar to other BNC corpus searching tools[1], which detail the source of the extract, the genre and dialect of speech. The added detail could prove useful for those researching idiom usage between differing sources. Another improvement to scale the range of use for researchers, would be to allow the choice to search through different English language corpora, for example, the Corpus of Contemporary American English (COCA). A successor to the current written BNC, (written BNC 2014) is also due for release to the public in 2020[27]. By comparing the two corpora, researchers will be able to shed light on how British English may have changed over the last two decades.

In order to support the new features, the Elasticsearch infrastructure could also be scaled to improve reliability and performance. The current Elasticsearch cluster uses a single node to keep within the AWS free tier usage, however, as mentioned previously, a big advantage to Elasticsearch is its ability to scale. A cluster can consist of multiple nodes. The index is then split across each node into shards. As each node contains a subset of documents, this allows parallel searches to be performed[28]. Replica nodes can also be configured, providing fault tolerance and higher availability[28].
To ensure security is maintained throughout the project's lifespan, a CI/CD pipeline could also be developed for project deployments. A CI/CD pipeline automates certain steps of deployment. This could be used to introduce tests to identify and update vulnerable packages in the source code before deployment. If the project were to be worked on by a team of developers, it would also be beneficial to ensure test coverage over the source code is increased. There are currently no automated tests within the React project, and the test-suite function tests are disjointed from the Python Lambda function tests, even though they are mostly the same. Adding better coverage of tests reduces the risk of unknown bugs occurring

during updates. It also provides a reasonable alternative to the use of extensive documentation over the codebase[29].

Reflection on Learning

I thoroughly enjoyed the process of developing this project and view the solution as a success. Working on this project allowed me to develop many new skills which will undoubtedly be beneficial in the future. Having only previously interacted with an Elasticsearch database using Kibana for checking logs (during my year in industry), I am pleased with the depth of implementation this project has allowed me to explore. It's not only taught me how to configure and control the low-level functionality of a search index but also provided me with a greater appreciation of the complexities that reside within these seemingly everyday tools. Learning the process of distributing the system using a cloud service has allowed me to conceptualise web application architecture in greater detail as well as inspired confidence in my ability to create future software solutions. Deconstructing the project architecture and source code during the security evaluation addressed many gaps in my knowledge for web security. Having implemented security measures in the system first-hand, gives me a greater appreciation for the range of threats that can exist, even for systems that don't deal with sensitive data.

Acknowledgements

I'd like to thank the project supervisor, Prof. Irena Spasic, for her support and guidance during the project. Also, thanks go out to Kathleen Beke, the owner of learn-english-today.com for allowing the use of her website’s idiom data.

References

[1]"British National Corpus (BNC)", *English-corpora.org*, 2020. [Online]. Available: https://www.english-corpora.org/bnc/. [Accessed: 07- May- 2020].

[2]L. Williams, C. Bannister, M. Arribas-Ayllon, A. Preece and I. Spasić, "The role of idioms in sentiment analysis", *Expert Systems with Applications*, vol. 42, no. 21, pp. 7375-7385, 2015. Available: 10.1016/j.eswa.2015.05.039.

[3]I. Spasic, L. Williams and A. Buerki, "Idiom—based features in sentiment analysis: Cutting the Gordian knot", *IEEE Transactions on Affective Computing*, pp. 1-1, 2019. Available: 10.1109/taffc.2017.2777842.

[4]G. Aston and L. Burnard, *The BNC handbook*. Edinburgh: Edinburgh University Press, 2008.

[5]A. Hume and D. Sunday, "Fast string searching", *Software: Practice and Experience*, vol. 21, no. 11, pp. 1221-1248, 1991. Available: 10.1002/spe.4380211105.

[6]Z. Galil, "On improving the worst case running time of the Boyer-Moore string matching algorithm", *Communications of the ACM*, vol. 22, no. 9, pp. 505-508, 1979. Available: 10.1145/359146.359148.

[7]M. Levene, *An introduction to search engines and Web navigation*. Hoboken: Wiley, 2010.

[8]"Elasticsearch: The Official Distributed Search & Analytics Engine | Elastic", *Elastic*, 2020. [Online]. Available: https://www.elastic.co/elasticsearch/. [Accessed: 07- May- 2020].

[9]"Install Elasticsearch with Docker | Elasticsearch Reference [7.6] | Elastic", *Elastic.co*, 2020. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html. [Accessed: 07- May- 2020].

[10]"AWS Free Tier", *Amazon Web Services, Inc.*, 2020. [Online]. Available: https://aws.amazon.com/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc. [Accessed: 07- May- 2020].

[11]"Security Best Practices in IAM - AWS Identity and Access Management", *Docs.aws.amazon.com*, 2020. [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html. [Accessed: 07- May- 2020].

[12]"Idioment homepage", *Users.cs.cf.ac.uk*, 2020. [Online]. Available: http://users.cs.cf.ac.uk/I.Spasic/idioment/. [Accessed: 07- May- 2020].

[13]"Index and search analysis | Elasticsearch Reference [7.6] | Elastic", *Elastic.co*, 2020. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-index-search-time.html#analysis-index-search-time. [Accessed: 07- May- 2020].

[14]"Porter Stemming Algorithm", *Tartarus.org*, 2020. [Online]. Available: https://tartarus.org/martin/PorterStemmer/. [Accessed: 07- May- 2020].

[15]"Match phrase query | Elasticsearch Reference [7.6] | Elastic", *Elastic.co*, 2020. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-match-query-phrase.html. [Accessed: 07- May- 2020].

[16]"Boolean query | Elasticsearch Reference [7.6] | Elastic", *Elastic.co*, 2020. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-bool-query.html. [Accessed: 07- May- 2020].

[17]"Highlighter breaks phrases · Issue #29561 · elastic/elasticsearch", *GitHub*, 2020. [Online]. Available: https://github.com/elastic/elasticsearch/issues/29561. [Accessed: 07- May- 2020].

[18]"Evergreen", *Evergreen.segment.com*, 2020. [Online]. Available: https://evergreen.segment.com. [Accessed: 07- May- 2020].

[19]"Introducing Hooks – React", *Reactjs.org*, 2020. [Online]. Available: https://reactjs.org/docs/hooks-intro.html. [Accessed: 07- May- 2020].

[20]"Host a Personal Website - Amazon Web Services (AWS)", *Amazon Web Services, Inc.*, 2020. [Online]. Available: https://aws.amazon.com/getting-started/hands-on/host-static-website/. [Accessed: 07- May- 2020].

[21]"Content Security Policy (CSP)", *MDN Web Docs*, 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. [Accessed: 07- May- 2020].

[22]A. Affairs, "System Usability Scale (SUS) | Usability.gov", *Usability.gov*, 2020. [Online]. Available: https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html. [Accessed: 07- May- 2020].

[23]J. Sauro, *Measuringu.com*, 2011. [Online]. Available: https://measuringu.com/sus/. [Accessed: 07- May- 2020].

[24]"Table of Contents | OWASP", *Owasp.org*, 2020. [Online]. Available: https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017. [Accessed: 07- May- 2020].

[25]"OWASP Serverless Top 10", *Owasp.org*, 2020. [Online]. Available: https://owasp.org/www-project-serverless-top-10/. [Accessed: 07- May- 2020].

[26]"Web Security", *Infosec.mozilla.org*, 2020. [Online]. Available: https://infosec.mozilla.org/guidelines/web_security. [Accessed: 07- May- 2020].

[27]"BNC2014 | ESRC Centre for Corpus Approaches to Social Science (CASS)", *Cass.lancs.ac.uk*, 2020. [Online]. Available: http://cass.lancs.ac.uk/bnc2014/. [Accessed: 07- May- 2020].

[28]R. Gheorghe, M. Hinman and R. Russo, *Elasticsearch in action*. Shelter Island, NY: Manning Publications Co., 2016.

[29]H. Percival, *Test-driven web development with Python*. Beijing: O'Reilly, 2014.

Appendix

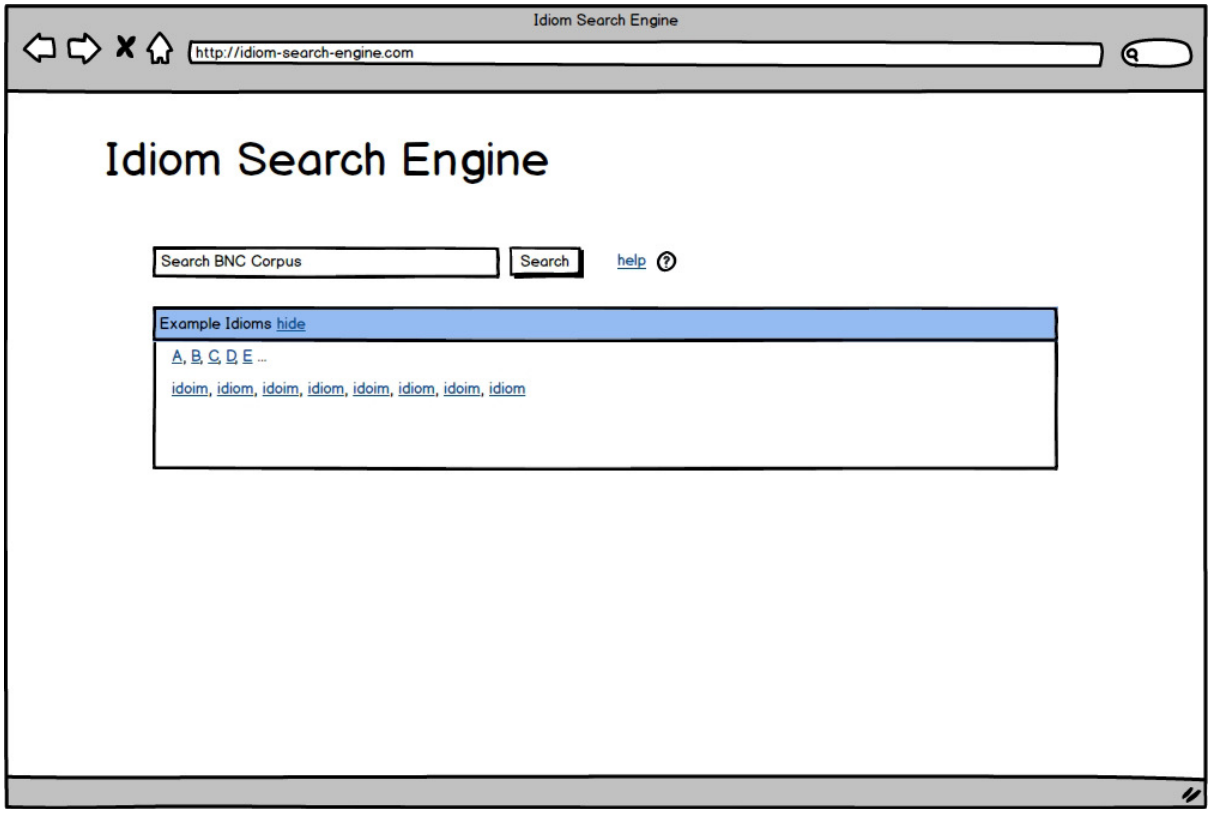

Figure 1: Mock-up design of the home page with the ‘example idioms’ box open.

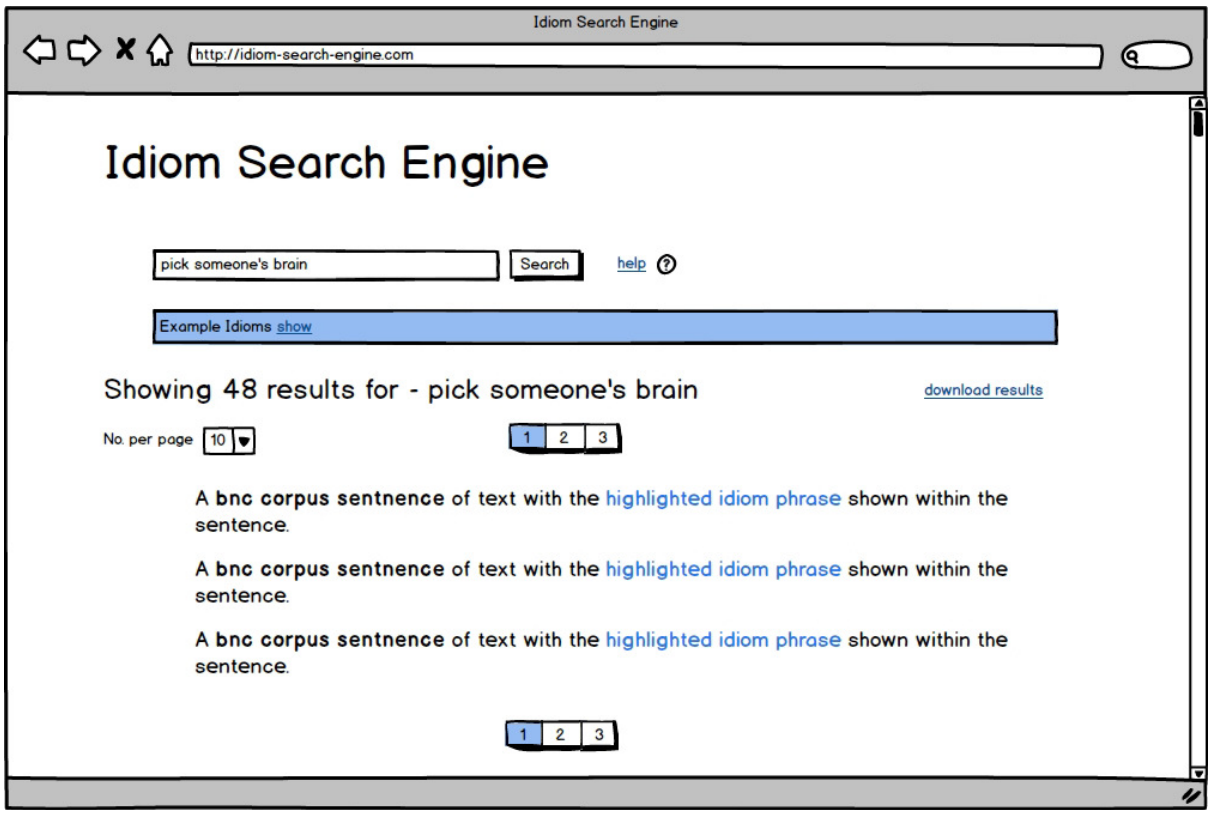

Figure 2: Mock-up design of the home page displaying results for a user search.

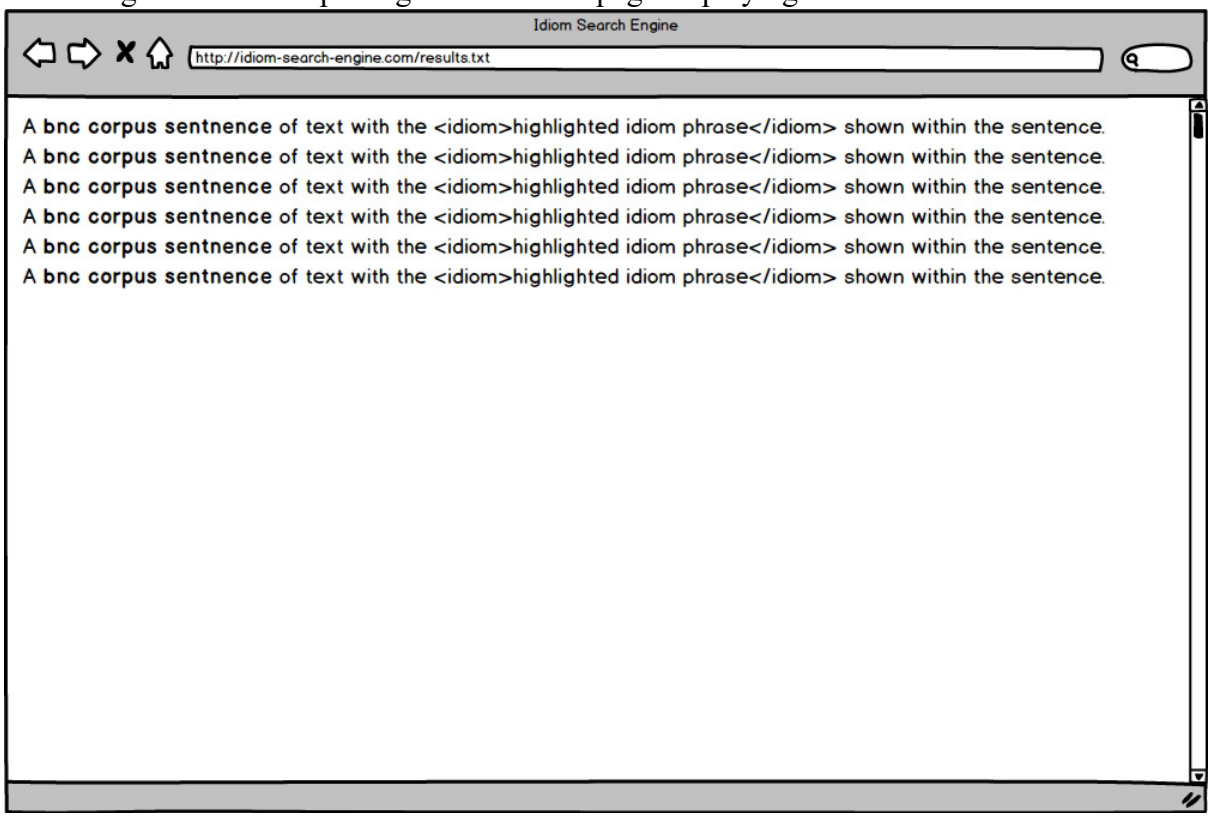

Figure 3: Mock-up design of the download results page showing the plaintext results of a search.

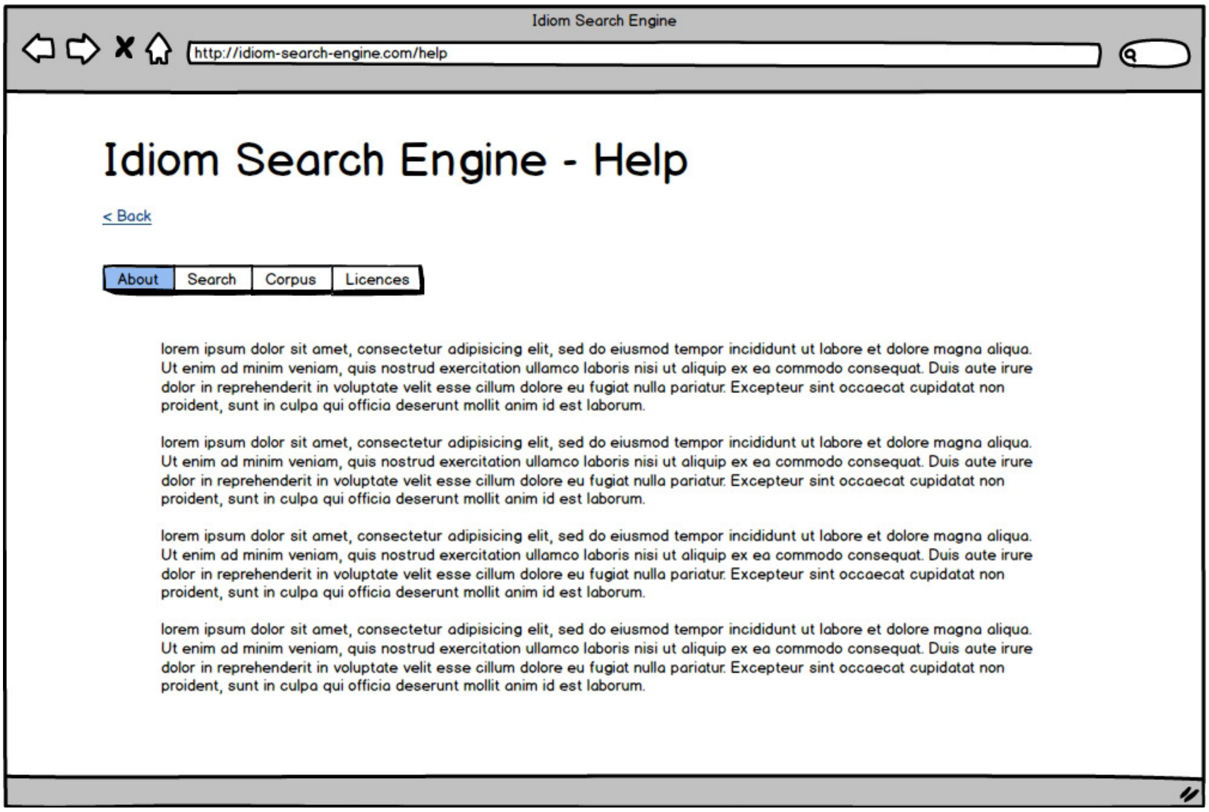

Figure 4: Mock-up design of the help page.

```
{
 "query":{
   "bool":{
     "should":[
       {
         "match_phrase":{
           "sentence":{
             "query":"call someone's bluff",
             "slop":4
             }
           }
         },
         {
           "match_phrase":{
             "sentence":{
               "query":"someone's bluff * call",
               "slop":5
             }
           }
         },
         {
           "match_phrase":{
             "sentence":{
               "query":"* bluff * call",
               "slop":6
             }
           }
         },
         {
```

```
            "match_phrase":{
              "sentence":{
                "query":"call * bluff",
                "slop":5
              }
            }
          }
        ]
      }
    },
    "highlight":{
      "fields":{
        "sentence":{
          "number_of_fragments":1,
          "fragment_size":1000,
          "pre_tags":"<idiom>",
          "post_tags":"</idiom>"
        }
      }
    },
    "size":10000
  }
```

Figure 5: A final processed query for the idiom ‘call someone’s bluff’ making use of all 4 possible ‘match_phrase’ clauses.