Northeastern University

CS6200 Final Project Report

Yik Lung Chan, Spring 2020 Semester

Project Definition

In this project I propose to build a general purpose financial news search engine that focuses on S&P 500 companies and news that might affect their stock prices. There are thousands of news publishers who publish news on the internet, and in this project I would build a pipeline to collect as many fresh news articles about S&P 500 companies as possible and provide a simple Google-like interface that allows users to query any news that are related to a specific company so that they can make informed decisions when trading in the stock market. Search results are guaranteed to be related to S&P 500 companies since only those news would be crawled and stored in the database.

Project Source Codes

Github Page

https://github.com/daviscyl/NewsSearch

Live Demo Website

http://ec2-54-219-141-55.us-west-1.compute.amazonaws.com:8080

Project Architecture

Overall this project is comprised of three main parts:

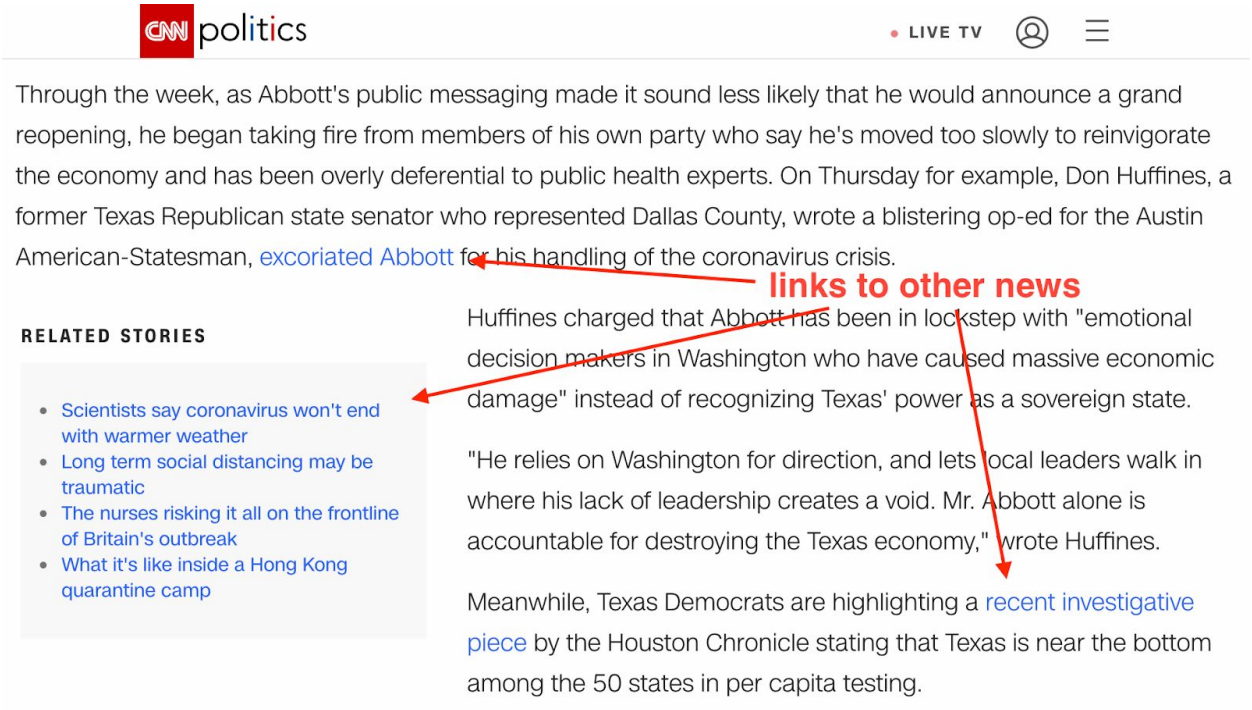
1. Data collection module, which collects as many news pieces related to financial sides of S&P 500 companies as possible, and sends these articles to the storage module. This module is also responsible to parse the news articles and extract title, url and url to cover images if available.
2. Storage + Search Engine module, which is responsible for storing the collected news articles and efficient search functionality.
3. Front End Website, which defines how the user will interact with the system and receive search results from the search engine.

Data Collection

Web Crawler

Since I want the news source to be reliable and not from websites with little trust-worthiness, I chose to use dedicated web crawlers on selected journalism websites with high credibility such as CNN, Wall Street Journal, Bloomberg, etc. One salient characteristic of news articles on these highly reputable websites is that they all reference other high quality news articles via embedded links and related stories sections.

The screenshot below from CNN website demonstrates this characteristics:



Sample CNN website news article

On average, one high quality news article would link to about 10 other news articles on the same website. This means we can start from a list of seed news articles and by following the links recursively, we can crawl nearly all news articles on the website by traversing the implicit graph of news articles (other than single articles with no links pointed to them). Since nearly all links are pointing to other news articles on the same website, crawling could be performed in a highly targeted and efficient fashion.

The actual crawling part of this project is done with the help of Python's *Scrapy*, *Splash*, and *Selenium* packages. During the crawling process I needed to prune the crawling links by identifying the real news article pages to speed up the crawling process since valid news articles often link to other valid news articles while other types of links rarely link to actual news articles even on the same website. This is done through verifying the link URL.

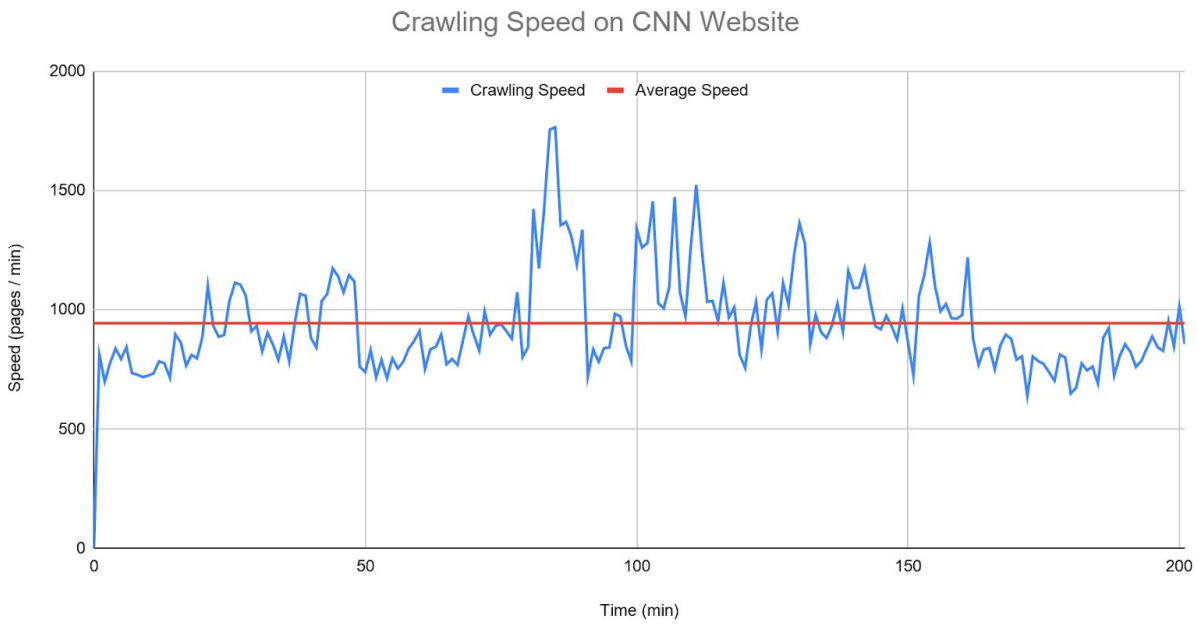
Let's take for example a couple valid news article on CNN website:
https://www.cnn.com/2020/04/17/politics/texas-economy-reopen-testing-coronavirus/index.html
https://www.cnn.com/2020/04/17/health/santa-clara-coronavirus-infections-study/index.html
https://www.cnn.com/2020/04/17/politics/quentin-hart-iowa-mayor-interview/index.html
https://www.cnn.com/2020/04/17/entertainment/sam-smith-album-name-change-intl-scli-gbr/index.html

We can see that any valid news articles on CNN website all have the *yyyy/mm/dd* date pattern in their URL following *cnn.com*. Leveraging this characteristic, I built a link extractor with pruning using regular expression that targets only URLs with *yyyy/mm/dd* date pattern in them and at

the same time avoiding video related contents. This is due to the fact that video web pages contain almost zero textual information and it is hard to extract the title information which is critical to the search engine's search functionality.

```
LinkExtractor(
        allow=r".*\d{4}\/\d{2}\/\d{2}\/",
        deny=("/videos/", "/video/", "cnn.com/interactive")
)
```

Though implementing the pruning rules, I was able to crawl over 154k valid news articles on CNN website in less than 4 hours. I extracted crawling speed data from the actual crawling logs, and plotted the data in the chart below. We can see that the crawling speed was quite stable with an average speed of about 940 pages per minute.



After the crawler downloads the website contents of a verified URL, the HTML parser takes the downloaded content (`response` in the following snippet), and uses XPath extractor to find the <h1> HTML content and use it as the title of the news article. This data is then sent to my ElasticSearch cluster for indexing and storage.

```
def parse_item(self, response):
    title = response.xpath("//h1/text()").get()
    if title:
        yield {
            "title": title,
            "url": response.url,
        }
```

Due to the time constraint, I was only able to make the crawler work on CNN website, and crawlers for other websites were not functioning well at the time of project demo. Therefore only the crawler for CNN website was included in the final project code repository.

Google News API

Another source of data that I used was the Google News API which provides enormous amounts of well-structured information from almost all credible sources on the planet. I used this API to gather targeted S&P 500 company related news from Google News' archive.

Specifically, I made a list of all S&P 500 companies and used the name of the company query and searched the API for top 100 news (limited by the free tier of News API) matching the company's name. The following code snippet demonstrates how this was done:

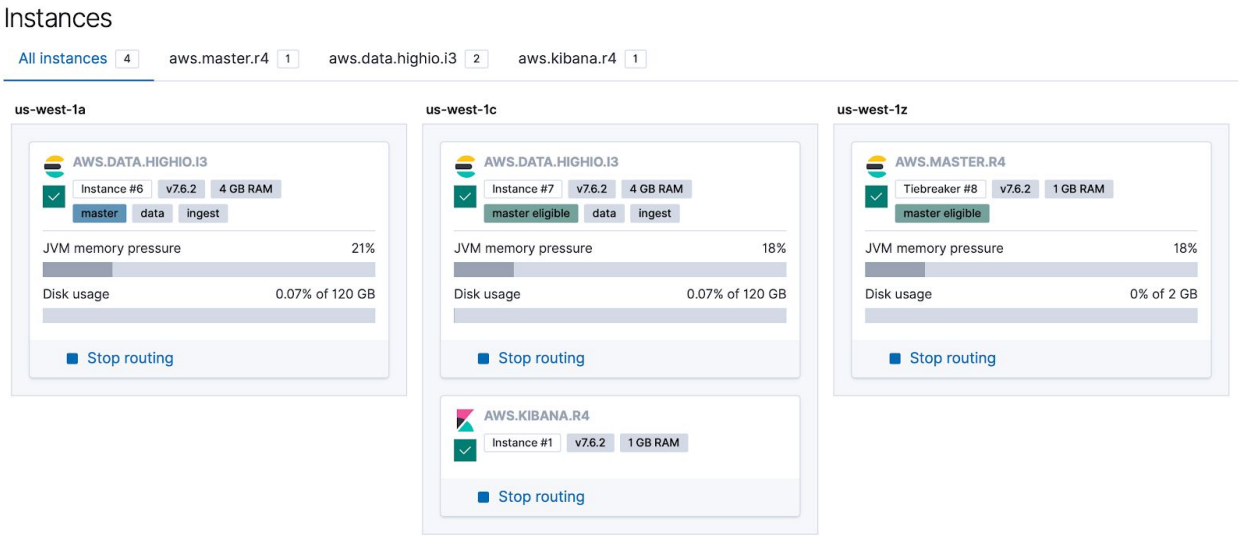
```
for i in range(len(companies)):
    news = apis[i % 2].get_everything(
        q=companies[i],
        from_param=day,
        page_size=100,
        language='en'
    )

    for article in news.get('articles'):
        es.index(index=index, body=article, id=article.get('url'))
```

Notice on the second line I used a list of API keys to perform the search (`apis[i % 2]`). This is also due to the limitation of the free tier of the News API which restricts API calls to 250 times per 12 hour period. I needed 500 searches for the 500 company names of the S&P 500 list, and I had to use 2 API keys in rotation to perform the full search.

Storage & Search Engine

Elastic Cloud Setup

My ElasticSearch cluster consists of a total of four nodes running on Amazon AWS EC2. Two of them are data nodes that contains identical copies of the sharded news webpage indexes; One node of the remaining two nodes is the master node which acts as a load balancer and routes search queries to the two data nodes for scalable search speeds; The last of the four nodes runs a Kibana server which me as the developer could easily monitor the ElasticSearch cluster and issue custom queries to help debug the system.



Indexing Process

Once a new piece of news article is sent to the ElasticSearch cluster for indexing, a standard process of pre-processing the data is performed as such:

1. English tokenizer tokenize the title (and description of the news if available)
2. Any punctuation or non-English letter character is removed
3. All texts are converted to lowercase
4. All texts are lemmatized to root form in English

Finally the lemmatized tokens of the news article are then added to the data nodes' inverted indices.

The search engine's inverted index is set up as a distributed, replicated, and sharded index cluster controlled by the master node. The inverted index has a replication factor of 2, and each of the two replicas consists of 24 shards of the index. When a search query hits the master node, it would use the map-reduce schema to divide up the job between the 2 data nodes by sending two sub-queries to them, and eventually merge the two half results. Then the list of results matching the query is sorted by BM25 match score between the title and the query before the final sorted list of results is sent back as response to the HTTP request.

This setup enables the cluster to grow horizontally and not lose search speed when the website traffic increases.

ElasticSearch API Proxy

Due to security reasons, Elastic Cloud does not allow developers to directly embed their project's username and password into front-end websites since all users of the website would have gained this secret information.

As a solution to this problem, I used another AWS EC2 machine as the API proxy that takes in queries from the front end website, and translates that query into ElasticSearch compatible query, then search my ElasticSearch database for news pieces whose title matches the query, and finally sends back the results via HTTP.

The proxy offers a simple API to the front end website where the string after the first '/' symbol in the url would be treated as the query. This setup was accomplished using Python's Flask package:

```
api.add_resource(ElasticsearchProxy, '/<string:query>')
```

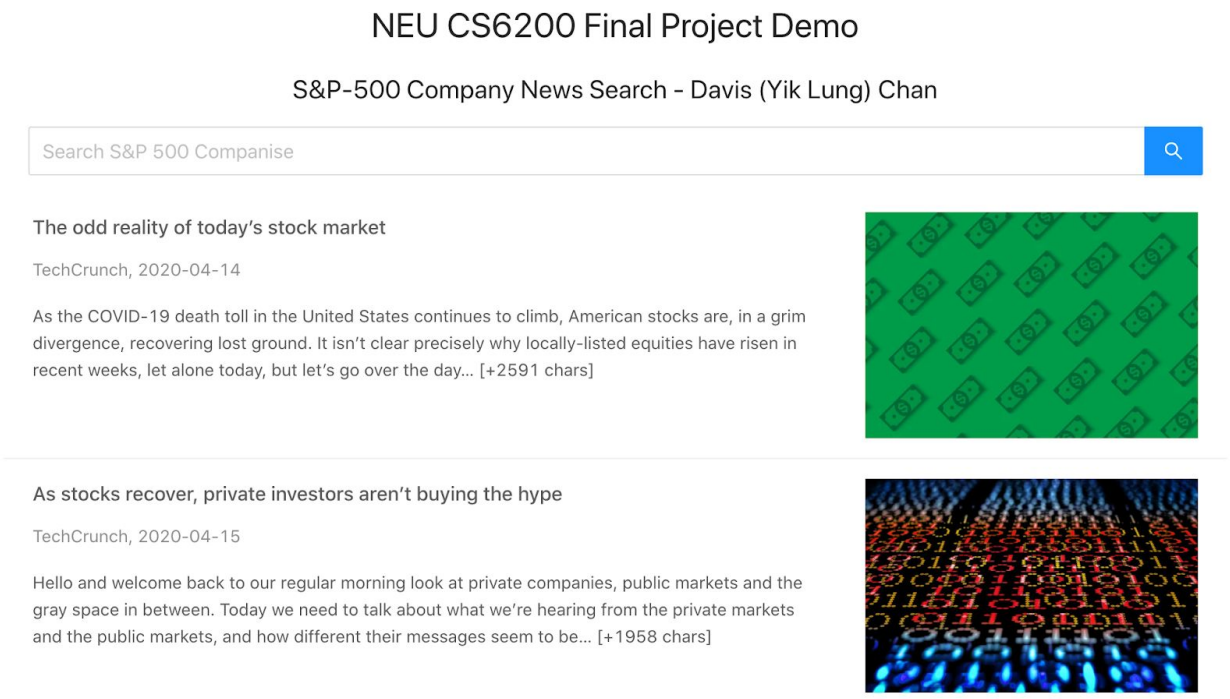
The main codes for query translation using ElasticCloud's Python client is pasted below:

```
def get(self, query):
    res = es.search(
        index='s&p500',
        body={
            "query": {"match": {"title": query}}
        },
        size=1000
    )
    return Response(
        json.dumps(res['hits']['hits']),
        headers={'Access-Control-Allow-Origin': '*'}
```

```
)
```

Notice I had to add to the response header "`'Access-Control-Allow-Origin': '*'`". This is done to get around the web browser's CORS(Cross-Origin Resource Sharing) regulation which generally does not allow a website to fetch resources from another domain. However AWS EC2's free tier does not have enough processing power to serve both the front end website and the API proxy on the same machine, and I had to use another machine to serve the API proxy. Adding the header lets the web browser know that fetching resources from another domain (a different IP address) is allowed by design and therefore does not block it.

Front End Website



Like the rest of the system, this web app's front end is also hosted on Amazon AWS EC2's free tier machine. When the user first visits the webpage, 20 pieces of today's highlight news is shown. When the user enters a query and hits the search button, a query is sent to the ElasticSearch API Proxy which returns a list of news information best matching the query to be rendered.

I'll cover the details of the front end design in this chapter.

Single-Page Web App via Reat.js

The front end of the search application is made using React.js which is a JavaScript package that enables easy module reuse compared to vanilla HTML+CSS websites. React.js allowed me to build the whole website as a single-page web app.

This means each user would need to request the website resource only once instead of requesting a new webpage every time the user hits the search button. The web app dynamically hides/shows elements of the web page according to the states of the web app, and uses internal routing (React Router) inside the browser locally to simulate the effect of going to a new address in the browser's address bar instead of actually sending a new request to the website server.

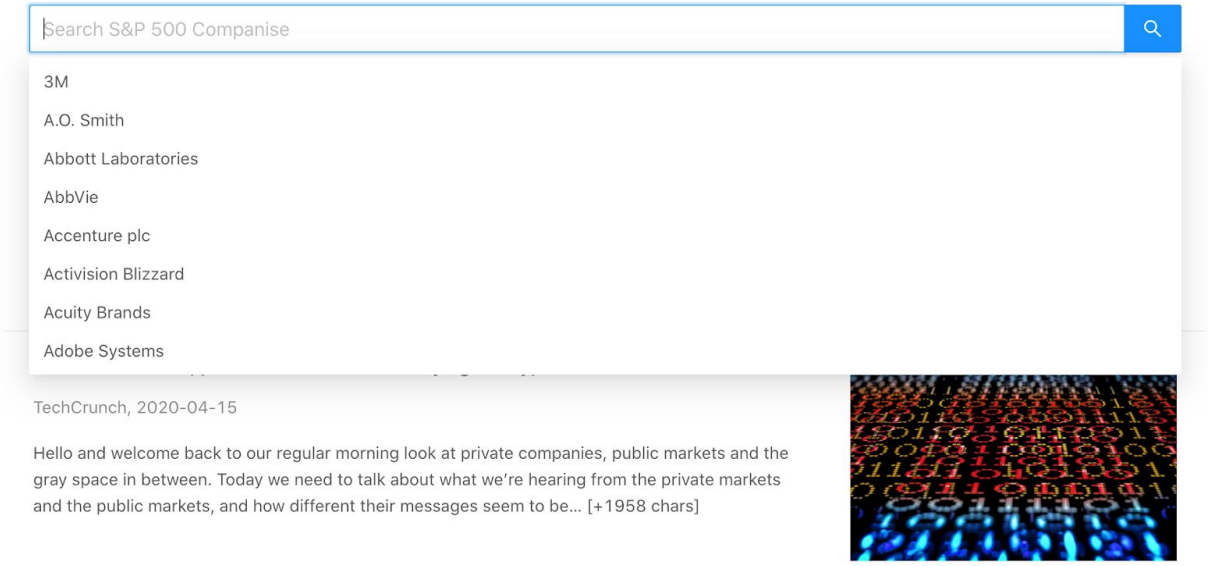
Since I'm only using the free tier machine on EC2 with limited processing power, the single page web design helps me stay within budget for this academic project.

Website Elements

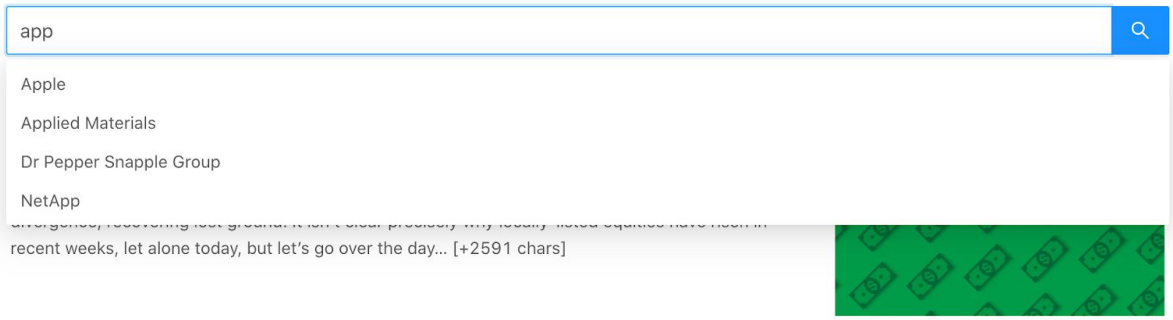
The elements of the website are based on AntDesign's React component templates which are offered to the public for free. I'll cover the main elements in the following sections.

Search Box With Auto-Completion

The main function of the website is to allow users to conveniently search. Therefore a search box is needed. I built this search box element with the auto-complete function built-in. Since I am building a website for searching company related financial news, the auto-complete function sources from a built-in list of the 500 companies listed in the S&P 500 market.



As the user types in the search box, the list of auto-complete candidates is filtered dynamically for the only company names matching the the input.

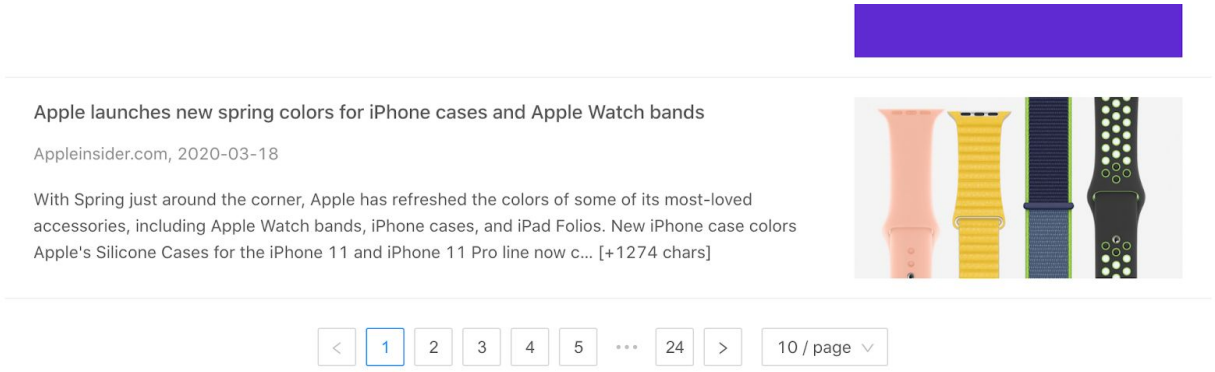


List View of News Results

Each piece of the results is rendered in 4 parts:

1. Title
2. Meta data including source & date
3. Short description clipped from the main article
4. Cover image

These elements are rendered with reasonable margins from each other and font sizes as seen in the screenshot below.

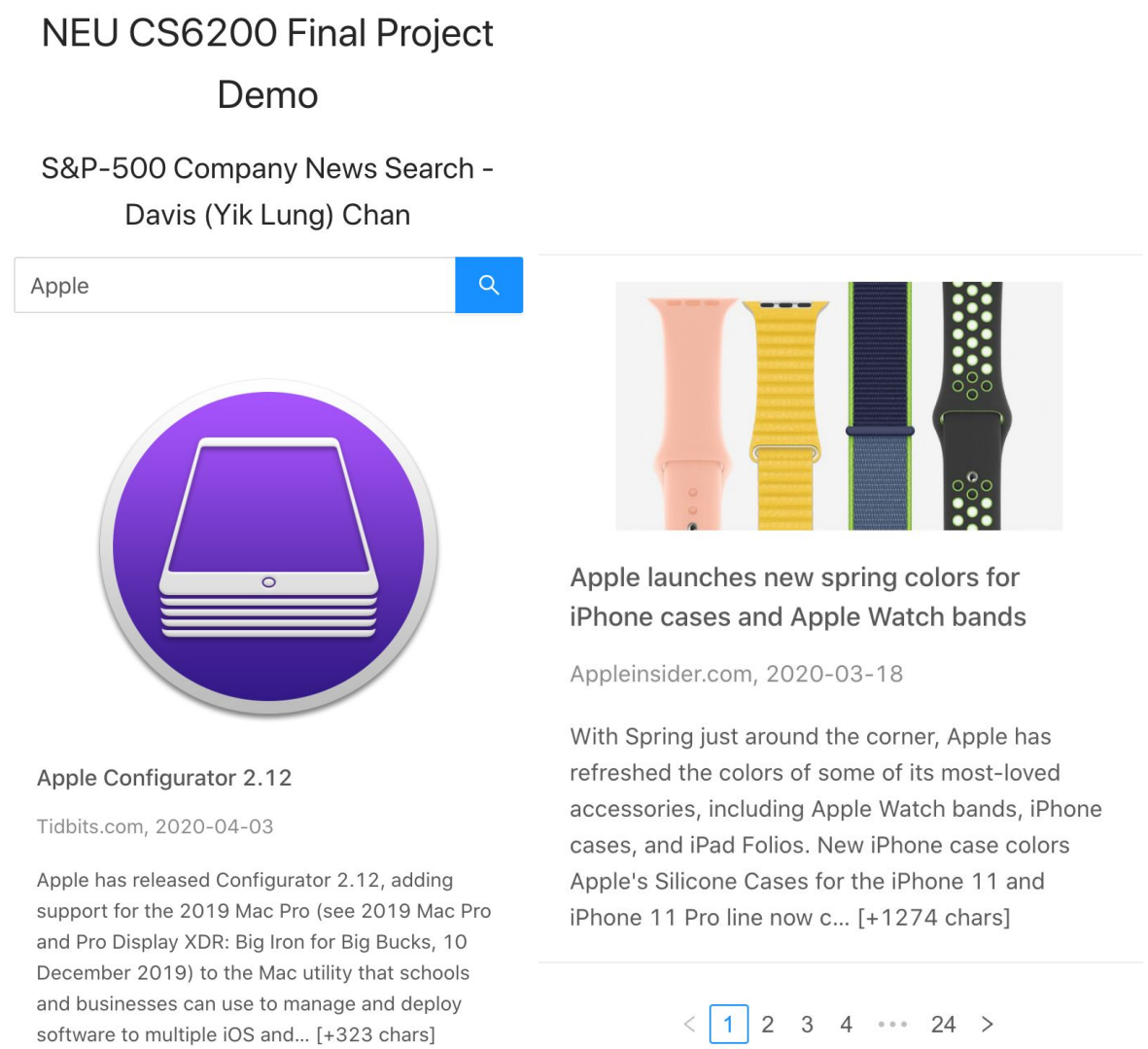


When the user performs searches, each query results can contain up to 1000 pieces of news articles matching the query. To handle the potentially large amount of data I used a pagination bar on the bottom of the web page which limits the browser to show only 10 pieces of news per page from the result list. This ensures fast and responsive user experience no matter the performance of the device the user is using.

Responsive Web Design

The website also incorporates responsive design to handle varying screen sizes on different devices. When the user visits the website from mobile devices, the website layout changes to accommodate to the smaller screen size as shown below.

Notice the layout of the news article is changed from horizontal to vertical, and the pagination bar at the bottom of the page is changed to a simple scheme to better fit small screens.



Future Improvements

News Crawler

In this project, I've only implemented one crawler that gathers news on one website. If time permits, I'd implement more dedicated crawlers for more credible sources of news. These news crawlers should be organized by a distributed system and controlled together by a central scheduler to fetch fresh news by the hour. This way the database would always stay fresh.

Search Matching Algorithm

In my ElasticSearch setup I only used inverted index on the query tokens which means if a user searches for "car companies", news with "automakers" in the title would not be considered a match. This can be improved by a better search algorithm that expands the search query with tokens that often appear near the tokens in the query. This way similar words would also be considered for ranking.

Customized User Experience

On my website, users can only search for company news they want to know at the moment they visit the website. If time permits, I should add a functionality that allows a user to log in to the website and view a customized website front page that uses a recommendation engine to automatically fetch headline news that's similar to the user's search queries in the past. Another way to improve the user experience is to allow users to subscribe to companies that he/she has invested in, and show only the news the user has subscribed to on the front page.