Personalized Bookmark Search Engine

CS 410 Text Information Systems (Fall 2021)
Prof. ChengXiang Zhai

Gazi Muhammad Samiul Hoque[*] **Yuheng Xie**[†] **Grace, Mu-Hui Yu**[‡] **Ying-Chen Lee**[§]

Team PBSE: Final Report

Contents

[*]ghoque2@illinois.edu
[†]yuhengx2@illinois.edu
[‡]muhuiyu2@illinois.edu
[§]yclee6@illinois.edu

Abstract

Most modern internet browsers, such as Google Chrome, now have a bookmark feature that allows users to save website URLs for future reference. If the bookmark list grows longer over time, users may find it difficult to access the relevant stuff they want. In this project, we create the personalized bookmark search engine as a Google Chrome browser extension with the idea of **Intelligent Browsing**, where we implement the "bookmark-and-search" feature, to address the searching difficulty and increase the accessibility of the bookmark feature. The plugin allows users to browse the web intelligently, and gives more accurate search results of relevant materials by going deeper into the website content rather than simply looking at the URL and title of a web page.

1 Introduction

In the context of the World Wide Web, almost all modern web browsers include bookmark[1] features. Bookmarks are called favorites or Internet shortcuts, and it could be normally accessed through a menu in the user's web browser. With bookmarking, users can save certain materials and accessible from anywhere for future reference. It seems like a terrific function for online browsers, however, it could be a little clunky as user's bookmark list grows larger over time. As a result, many external programs, in addition to bookmarking options within most browsers, provide bookmark management. Most of the bookmark management solutions, on the other hand, are solely concerned with storage and need the user to manually organize the content. Without considering the future accessibility to the bookmarked items, finding the needed content from a huge bookmark collection could be a nightmare. Furthermore, judging merely by the URL and web page titles of a particular bookmarked item can make it difficult to retrieve the relevant material from the bookmark list.

Therefore, we propose a Personalized Bookmark Search Engine to address these issues. Users may simply save the current page to their bookmark list while surfing with this plugin, and receive relevant search results by entering target keywords. For easy installation, we will develop the search engine as a Google Chrome browser extension. When the user bookmarks a website using our extension, it will send the current web page address to our back-end server. There it will parse and index the page for the user. A search box will also be implemented in the widget that allows the user to enter a query term and it will fetch relevant contents from the index through our back-end API.

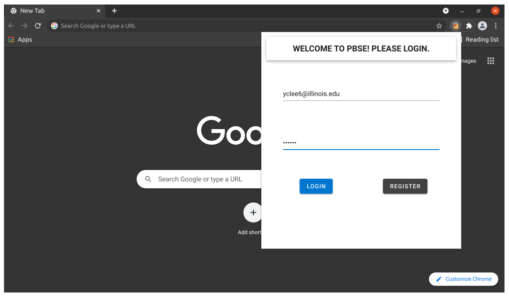
The theme of our project is **Intelligent Browsing**, where we implement the "bookmark-and-search" function on top of Google Chrome to allow users to browse intelligently through the web. We'll apply the web crawling, indexing, and searching techniques we have learnt in this class for this project.

2 Implementation

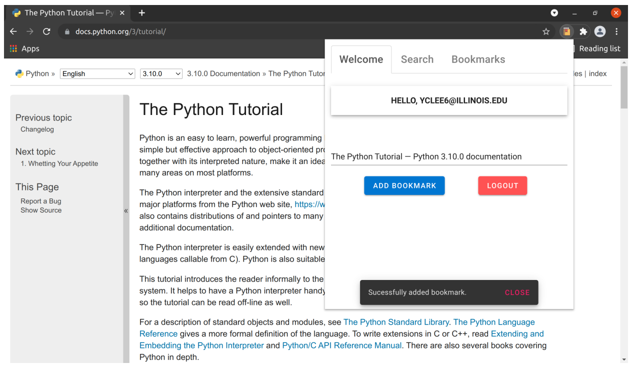
Figure 1 shows the GUI of our bookmark search engine and demonstrates how to use this tool. After registering an account using an email, users can log in to our Google Chrome extension and use the services. Users can save the current website as a new bookmark. After adding a new bookmark, the bookmark will be shown in the bookmark list. When users want to search for any specific topics over the bookmarked contents, they can make a query by giving keywords, and the bookmarks that match the query would be given in a list. The matching score of each returned bookmark would be given as well.

In this section, we are going to discuss the architecture and demonstrate how our project was implemented. First, we will briefly show how the components are connected, and then we will look into details of how our backend and frontend were implemented.

[1] Bookmark (digital): `https://en.wikipedia.org/wiki/Bookmark_(digital)`

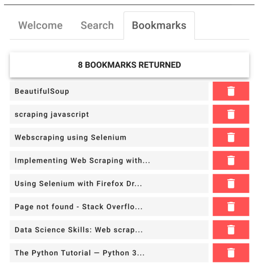


(a) login


Welcome
Search
Bookmarks
HELLO, YCLEE6@ILLINOIS.EDU
The Python Tutorial — Python 3.10.0 documentation
ADD BOOKMARK
LOGOUT
CLOSE

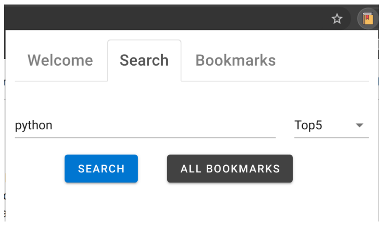
Sucessfully added bookmark.

(b) add bookmark

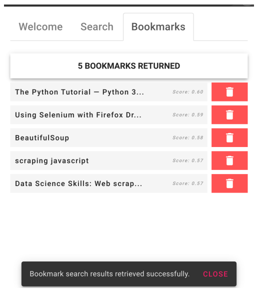


BeautifulSoup
scraping javascript
Webscraping using Selenium
Implementing Web Scraping with...
Using Selenium with Firefox Dr...
Page not found - Stack Overflo...
Data Science Skills: Web scrap...
The Python Tutorial — Python 3...

(c) all bookmarks



(d) search bookmark



The Python Tutorial — Python 3...
Using Selenium with Firefox Dr...
BeautifulSoup
scraping javascript
Data Science Skills: Web scrap...

Bookmark search results retrieved successfully.

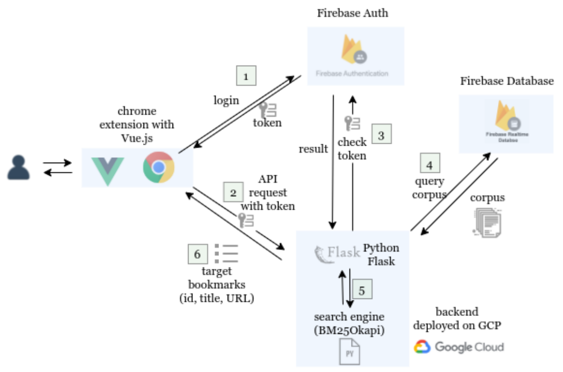
(e) search results

Figure 1: GUI of our chrome extension

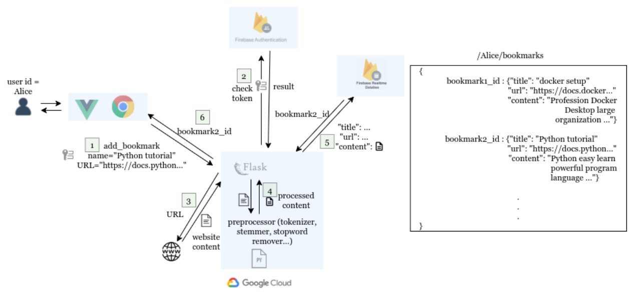
2.1 Architecture

Our project supports features such as login, add a new bookmark, show all bookmarks, search bookmarks, and delete bookmark. To achieve this, we have the frontend implemented in Chrome extension with Vue.js, the backend implemented with Python Flask, which is deployed on Cloud Run from Google Cloud Platform (GCP), and Firebase services helping us authenticate users and store the contents of users' bookmarks.

Figure 2 shows the architecture diagram of our project. It also briefly shows how login, searching bookmarks, and adding bookmarks are handled. In this subsection, we will briefly show how the components are connected to each other, and their roles in this project.



(a) login and search bookmarks



```
{
    bookmark1_id : {"title": "docker setup"
                    "url": "https://docs.docker..."
                    "content": "Profession Docker
                               Desktop large
                               organization ..."}

    bookmark2_id : {"title": "Python tutorial"
                    "url": "https://docs.python..."
                    "content": "Python easy learn
                               powerful program
                               language ..."}
                    .
                    .
                    .
}
```

(b) add bookmark

Figure 2: Architecture diagram

2.1.1 User Management

We used Firebase Authentication[2] to authenticate users of our Chrome extension. When users log in, our frontend retrieves an ID Token for the user using Firebase Auth SDK. This ID Token serves as an authorization key for the user to use the services offered by our backend. Thus, after the frontend get the Token, it attach it in the header of HTTP request and send this request to our backend. The backend authenticates a particular user by validating the token and authorizes for further API requests using Firebase SDK. If the token is invalid, then an error is returned from the backend, otherwise the request is processed.

2.1.2 Communication between Frontend and Backend

Our backend is the server that accept HTTP requests. It exposes several APIs (see Appendix B) for the frontend to access to. When receiving request from users, the frontend sends corresponding HTTP request with Token attached on it. After the backend finishes the request, it sends the response back to the frontend. The frontend will then parse the response and show the results in GUI (see section 2.3 for details).

2.1.3 Bookmark Management and Bookmark Retrieval

In order to respond to the request, the backend needs to pre-process or post-process the content of bookmarks, such as crawl the web pages, stem and tokenize the web page content, or search bookmarks using BM25 algorithm. All these tasks are done in the backend. However, if we store all the processed contents of bookmarks in the backend, it would be inefficient and unscalable. Using a well-designed cloud database management system might be a better solution.

In this project, we used the Firebase Realtime Database[3] to manage the contents of bookmarks. We used the Firebase SDK[4] in our Python scripts, and then we can easily use the SDK to store data into or retrieve data from the database. An excerpt from the code is shown as below:

Store data in Firebase Realtime database

```python
from firebase_admin import db
...
ref = db.reference("/"+user_id+"/bookmarks/")
content = {
    "title": bookmark_title,
    "url": bookmark_url,
    "content": parsed_content,
}
p = ref.push(content)
new_bookmark_id = p.key
...
```

Query data from Firebase Realtime database

```python
from firebase_admin import db
...
ref = db.reference("/"+user_id+"/bookmarks/")
bookmarks = ref.get()
...
```

There's two noteworthy points in these code blocks. First, when adding a new bookmark, the title (bookmark's name given by the user), the URL of the bookmark, and the processed (stemming, tokenizing, etc.) content of the web page are stored in the database. Second, after the bookmark is added, the backend will get a key for this bookmark object. This key is just like the key for Python's dictionary, and it serves

[2] `https://firebase.google.com/docs/auth`
[3] `https://firebase.google.com/docs/database`
[4] `https://firebase.google.com/docs/reference/admin/python/firebase_admin.db`

as the bookmark's identification. Under `/{USER_ID}/bookmarks/`, a dictionary is stored. Each item in this dictionary is a bookmark object. Therefore, after getting this dictionary, we can get the target bookmark with the bookmark's identification.

2.2 Backend Implementation

After knowing the interaction between the components, it's important to know how the web page is crawled, how the web page is processed, and how the search engine is implemented. All of these important functions are stored in the backend. In this section, we will show how they were implemented.

2.2.1 Crawling of Web Pages

The `crawl(page_url)` function is in `/backend/contents.py`. Given a website URL, this function crawls that web page with a Python package called `bs4.BeautifulSoup`. `html.parser` of BeautifulSoup is used to parse the website. The text of the HTML page would be retrieved. For now, our crawler doesn't support dynamic websites using JavaScript.

2.2.2 Pre-processing for the Content of Web Pages

The pre-processing functions are in `/backend/utilities.py`. After the text of the website is crawled by our `crawl(page_url)`, we pre-process the content using `tokenize_str(text)` function. This function has performs the following steps:

1. `remove_multiple_spaces(text.lower())`: It first removes all additional whitespace characters from the text, as well as make the content lowecase.
2. `remove_stopwords(text)`: Next, we remove all stopwords[5] from the content.
3. `word_tokenize(text)`: We then tokenize the full content and return a list of words.
4. `remove_special_characters(words)`: We remove every special characters (such as HTML tags) from the words.
5. `stemming(words)`: Finally we do stemming on each words using Porter Stemming[6].

Several Python package are used to do these processing. `gensim` is used to remove stopwords, and `nltk` is used to tokenize the words and stem the words. These pre-processing functions are not only used for text of web pages, but also for query terms given by users; otherwise, the query terms might not be able to match the processed words in the text.

2.2.3 Bookmark Search Engine

The search request is processed using the `search_query(query, top_n=5, corpus=[])` function, which is in `/backend/search.py`. It get the scores of the query with the BM25Okapi algorithm[7] implemented in the Python package called `rank_bm25`[8]. Given a query $Q = q_1 q_2 \dots q_n$, the BM25 score of a document d is defined as follows:

$$\begin{aligned} score(d,Q) &= \sum_{i=1}^{n} IDF(q_i) \frac{c(q_i,d)\cdot(k_1+1)}{c(q_i,d)+k_1\cdot\left(1-b+b\cdot\frac{|d|}{avgdl}\right)} \\ IDF(q_i) &= ln\left(\frac{|C|-df(q_i)+0.5}{df(q_i)+0.5}\right) \end{aligned}$$

[5] What are Stop Words?: https://www.opinosis-analytics.com/knowledge-base/stop-words-explained/
[6] The Porter Stemming Algorithm: https://tartarus.org/martin/PorterStemmer/
[7] https://en.wikipedia.org/wiki/Okapi_BM25
[8] https://github.com/dorianbrown/rank_bm25

where $c(q_i, d)$ denotes the word count of q_i in document d, k_1 and b are free parameters, $|d|$ denotes the length of the document d, $avgdl$ is the average document length in the corpurs, $|C|$ is the number of documents in the corpus C, $df(q_i)$ is the number of documents that contains the term q_i.

After all the scores of the documents in the corpus were calculated, the first top n results would be returned by the function `search_query(query, top_n=5, corpus=[])`.

2.3 Frontend Implementation

The frontend chrome extension is written in Javascript and it compromises of many GUI components bootstrapped using the VueJS framework. The user can interact with these components to perform various functions such as logging in, adding bookmarks and searching on bookmark contents. The chrome extension definitions file, `manifest.json`, contains the configurations needed to initialize the application. The background script, `manifest.json` exists permanently during the lifetime of the extension and it takes of user authentication using Firebase. Firebase and backend endpoint configurations are stored in `config.json`. The popup page, `popup.html` routes to `App.vue`, which contains the main HTML GUI templates and bookmarking functions such as `addBookmark()` and `searchBookmarks()`. These bookmarking functions send requests to backend endpoints using REST API calls, and they receive the relevant responses after that, which will be updated in the GUI.

2.4 Deployment

We deployed our backend in Cloud Run (Google Cloud Platform) as a dockerized container. We created a project in GCP and enabled the Cloud Run service. We have also created a `Dockerfile` to use for deploying in Cloud Run[9]. By leveraging the Cloud Run and Firebase, our app is truly serverless, autoscaled and load-balanced by default.

3 Evaluation

We used both user-based (Cranfield evaluation) and search-engine-based methodologies for the evaluation.

3.1 User-based evaluation

In the user-based evaluation method, five users contributed to the annotation of our testing data. Each user submitted a list of 15 bookmarked websites, three query keywords, and the human-labelled relevance score of each website for each query keyword. The score can be 1-4 or 1-5, it depends on the users' preference. Also, they were asked to give top5 results for each query keyword.

3.1.1 Top 5 Normalized Discounted Cumulative Gain (nDCG@5)

The following table shows the nDCG@5 of the results.

	User 1	User 2	User 3	User 4	User 5
nDCG of the 1st query	0.98	0.98	0.90	0.95	1
nDCG of the 2nd query	0.95	0.82	0.98	0.98	0.83
nDCG of the 3rd query	0.98	0.73	0.85	0.86	0.77

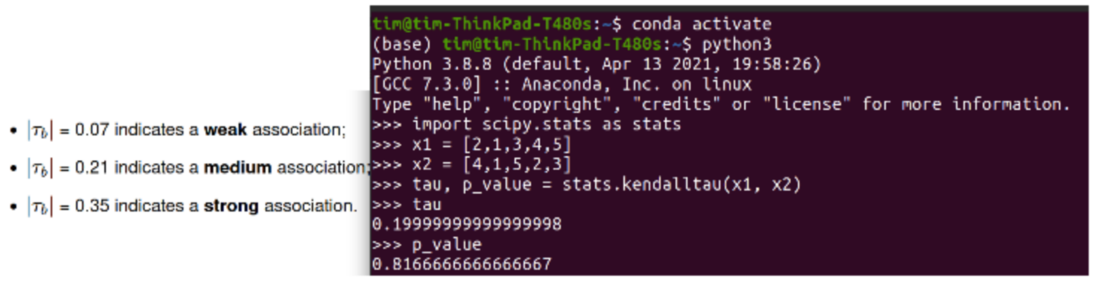
Even though nDCG is a method being used to compare different Text Retrieval system, not to evaluate a single system, we still can use this criteria to roughly see the behavior of our system.

Based on our own experience, we found that if the nDCG is greater than **0.9**, the searching results are satisfying. From the experiment results, we found that nearly 1/3 of the results are not satisfying. After examining the websites, we found that the results are poor when the web pages are not text-based. If the users only store text-based web pages, the results are quite satisfying.

[9] Build and deploy a Python service: `https://cloud.google.com/run/docs/quickstarts/build-and-deploy/python`

3.1.2 Kendall Tau

Kendall Tau is a criteria to measure whether two ranking list are correlated to each other[10]. Figure 3 shows the Kendall Tau value of two top5 ranking lists. However, the $p - value$ is too high, which indicates that this result is not statistically significant. It may because of that comparing top5 ranking is not enough. Maybe we need to compare top10 or more. Since it cannot provide us meaningful evaluation, we didn't use this method to evaluate all the experiment results.



- $|\tau_b|$ = 0.07 indicates a **weak** association;
- $|\tau_b|$ = 0.21 indicates a **medium** association;
- $|\tau_b|$ = 0.35 indicates a **strong** association.

```
tim@tim-ThinkPad-T480s:~$ conda activate
(base) tim@tim-ThinkPad-T480s:~$ python3
Python 3.8.8 (default, Apr 13 2021, 19:58:26)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy.stats as stats
>>> x1 = [2,1,3,4,5]
>>> x2 = [4,1,5,2,3]
>>> tau, p_value = stats.kendalltau(x1, x2)
>>> tau
0.19999999999999998
>>> p_value
0.8166666666666667
```

Figure 3: Results of using Kendall Tau

.

3.2 Search-Engine-based Evaluation

In the search engine-based evaluation approach, We used three query terms on Google Search and retrieved the top five results for each query keyword, and the top 15 websites were then added to our bookmark tool. The top five results for each of the three query terms were then retrieved and compared with the Google results.

We conducted the testing twice. The three query terms for the first set are "natural language processing, deep learning, data structure", which are relatively different domains; while the three query terms for the second set are "Christmas traditions, Christmas movies, Christmas present", which are considered more similar and relevant than the first set.

The following table shows the nDCG@5 of the results.

	Set 1	Set 2
nDCG of the 1st query	0.98	0.98
nDCG of the 2nd query	0.99	0.90
nDCG of the 3rd query	0.99	0.87

When comparing the two testing sets, we discovered that the model performs better and is more stable on the set with more distinct concepts, however the top 1 or 2 results on the second set, which contains some concepts overlapped, still work well, but it may introduce noise in the remaining items.

4 Discussion and Future Work

Due to a shortage of time, many various adjustments, testing, and experiments have been postponed (i.e. the experiments with real data collected from users are usually very time consuming in terms of recruiting and data labelling). During the review process, we discovered various insights and items, and future works will focus on a more in-depth analysis of particular systems, new recommendations to test new techniques, or simply curiosity. There are a few suggestions that we would have liked to try in relation to the design of the scoring model mentioned in Section 2.2.3 and the evaluation methods presented in Section 3:

[10] kendalltau: `https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.stats.kendalltau.html`

1. When designing a web page, the web page designers often name the section headings by using keywords. We can take advantage of this to optimize our system. For example, we can give higher weights on the words that appear as section headings when crawling the page by repeating those words in the HTML elements such as h1, h2, or h3. In this way, these words have higher word counts in the content, and therefore would be given higher score by the BM25 algorithm.

2. We found that the results are not acceptable when the web pages are not text-based. They might not have those keywords inside, but those keywords are actually highly related to the web pages. For example, Google Scholar[11] is strongly related to the keyword "research paper"; however, you cannot get any "research" or "paper" in it. In this kind of condition, our search engine gives bad results.

   To deal with this problem, we can search the title of the bookmark using Google search engine and crawl some of the searching results[12]. For example, when our system receives a request to add Google Scholar into the bookmark list, besides crawling that web page, our systems can use the Google search engine to search "Google Scholar" and crawl the top five searching results. These results would probably contains the descriptions on what Google Scholar is, such as "Google Scholar is a tool for searching research paper". In this way, "research paper" can be included into the content.

3. As shown in Section 3, we used normalized Discounted Cumulative Gain (nDCG) and Kendall tau to evaluate our system. However, these might not be applicable to our case.

   To make the evaluation results more convincing, we need to find other ways to do this. We can compare our nDCG with another well-design TR system. Moreover, we can find some ways to take advantage of the scores given by our system. Those scores actually serve as good indicators of the researching results, but we haven't found a way to take advantage of it.

4. In our search-engine evaluation study, we discovered that employing a group of similar query terms can cause additional noise in the search results. However, we spoke with three users and discovered that they actually enjoy the extra information that appears on the result list on occasion. One of the causes is that individuals are sometimes unsure of what they actually want, making it difficult for them to formulate a specific query. Another factor is that different content tends to distract certain users (and they like it), so this may differ depending on the preferences of individual users. One user, on the other hand, finds it extremely irritating and just wants to read the most relevant stuff as soon as possible. To create a bookmark tool with better user experience, we can allow users choose how much noise they want in their search results. Also, how Google performs in terms of page ranking could also serve as a guide for us in the future, such as page linking, search history, time and location context, etc.

5 Team Contributions

All of our team members have spent more than 20 hours on the project individually, and we have completed our individual tasks successfully. We all have participated in others' tasks through discussions and by pair-programming. Below are brief contributions by the respective members of our team:

Gazi Muhammad Samiul Hoque (NetID: `ghoque2`**) - Captain**

Developed the API architecture, managed the Google Cloud Project, Integration and Deployment using Docker and Google Cloud Build

Yuheng Xie (NetID: `yuhengx2`**)**

Designed and developed the frontend GUI and Javascript functions, managed the Firebase authentication service and integrated the frontend with backend

[11] `https://scholar.google.com/`
[12] Crawling Google Search Results: `https://dev.to/samzhangjy/crawling-google-search-results-1c9d`

Grace, Mu-Hui Yu (NetID: `muhuiyu2`)

Implemented the crawling and indexing functionalities, backend API development and testing

Ying-Chen Lee (NetID: `yclee6`)

Developed and assessed different evaluation systems, backend API testing and integration

6 Conclusion

In this project, we tried to develop a personalized bookmark search engine as a Google Chrome browser extension, where we implemented the "bookmark-and-search" function to enhance the bookmark feature's accessibility. This plugin helps users explore the web intelligently and provides more accurate search results of relevant resources.

The system is evaluated using both user-based (Cranfield evaluation) and search engine-based (Google search) methodologies. After examining the evaluation results and the given web pages, we found that the system is more effective when most of the web pages are text-based. If not, the results are less effective.

Appendix

A Artifacts

A.1 Code Repository

Our project's code repository is hosted at `https://github.com/samhq/CourseProject`. You can review and try the implementation at your own by forking the repository. The installation instructions are given at the `README.md` (`https://github.com/samhq/CourseProject/blob/main/README.md`) file.

A.2 Tutorial Presentation

We have uploaded our tutorial presentation to UIUC Box. You may view it at `https://uofi.box.com/s/f5fd5acnr0edicrc94d207l9zu7peh8f`.

B API Documentation

B.1 Primary API

`/get_all_bookmarks`, ***method*=GET**

Params None

Header `'Authorization': 'USER_TOKEN'`

Response A dictionary of bookmarks of the user

`/search_bookmark`, ***method*=POST**

Params `query`: *Keywords for searching*, `top_n`: *Number of items to return*

Header `'Authorization': 'USER_TOKEN'`

Response A dictionary of searched bookmarks with score

`/add_bookmark`, ***method*=POST**

Params `bookmark_name`: *Web page title*, `webpage_url`: *Web page URL*

Header `'Authorization': 'USER_TOKEN'`

Response `bookmark_id` *if successful*, `error` otherwise

`/delete_bookmark`, ***method*=POST**

Params `bookmark_id`: *ID of the bookmark*

Header `'Authorization': 'USER_TOKEN'`

Response `error = False` *if successful*, `error = True` otherwise

B.2 Helper API

`/api/userinfo`, ***method*=GET|POST**

Params None

Header `'Authorization': 'USER_TOKEN'`

Response The `user_id` from Firebase DB

`/api/signup`, ***method*=POST**

Params `email`: *User Email*, `password`: *User's chosen password*

Header None

Response `error = False` *if successful with `user_id`*, `error = True` otherwise

`/api/token`, ***method*=POST**

Params `email`: *User Email*, `password`: *User password*

Header None

Response `token` *JWT token if successful*, `error = True` otherwise

B.3 Middle-ware

`check_token()`

It check for `Authorization` header. If present, it verifies the token, get the user back from Firebase and sets a variable `request.user = user`. Then it sends the control back to the API to process further. If it cannot authenticate the user, or the authorization token is not present (or invalid), it terminates the request with the `error` message.