URL SHORTY - i253 Final Project Technical Report

Tiffany Barkley, Chris Fan, Faith Hutchinson, & Cameron Reed

READ ME

- Our URL Shorty site is available online at http://people.ischool.berkeley.edu/~chrisfan/server/register
- To run Flask, execute “app_combine.py” from a HTTPS accessible location on the webserver.

Overview

urlShorty is an application that lets users shorten long URLs and manage short and long URL associations over time. It allows users to define custom short names for their URLs, and also contains mechanisms for generating random URLs as well as suggested URLs based on the URL’s html title tag. This Overview describes our work on the following four features:

1. Secure Username Logon
2. Customized URL Shortening Services
3. Mobile Responsive Design
4. Recommended Short URLs based on Selected Website

Part 1: Secure Username Logon

User Authentication: Users can sign up with a username and password, then log-in in subsequent sessions to view URLs they have already created. We store usernames and passwords in a table of our database. When a new user attempts to create an account, the database is checked to confirm that the user name does not exist. If the user name does exist, then we inform the user that the user name is not available. Given the user selects an available user name, we then store the username and password in our database. Using bcrypt, we encrypt password the user enters and store the secured password data in our user database. We also use bcrypt to securely verify the user’s plaintext password input with her encrypted password.

The application is made stateful through the use of the Flask session module. The username is then stored as a key-value pair in the session object (session[‘username’] = encrypted username) that is sent to the browser as a session cookie. The server then makes sure that the browser sends back this cookie with each request before retrieving the results. A log_in required decorator was used to make sure that users are logged-in before they can access certain pages. The decorator checks to see if the browser sent a valid username cookie. If it did, the get request is served. If not, the user is redirected to the log-in page. The user can log-out from any page that requires authentication. Upon clicking the logout link, the server clears the session object and redirects the user to the log-in page. With the logout link, the server clears the session object and redirects the user to the log-in page.

In the process of learning about encryption, sessions , and other security issues, we realized that we could have added another layer of confidentiality within our database function file. At present, our Flask file imports database functions from a Python file, db_functions.py. We currently initiate a connection to the database with a function in db_functions.py, and in so doing, we reveal sensitive information about the login credentials to our database. One way we would have addressed this is to store the credentials in a secure file -- a file containing access restrictions and perhaps encryption--which would then be read into db_functions.py. However, for the present project, we have decided to rely on the application files remaining in a secure section of the server.

When hosting, urlShorty should be run on a webserver supporting HTTPS for clients to send information over SSL.

Part 2: Customized URL Shortening Services
Our application provides for the storage of URLs by user. In order to enable users to preserve short URLs they have created, we had to ensure that previously created short URLs remain persistent. This requires querying the database of links each time a user attempts to create a new URL. A select query is sent to the database for all short URLs, and if the short URL input by the user already exists, then the user is notified of the conflict and instructed to either try one of the suggested links or enter a new short URL. The main complication was ensuring that we implemented this checking (done via ajax calls to the server) at the appropriate times. For example, we check to make sure each of the randomly generated URLs is available before displaying it to the user for submission. We also check all of the Twitter suggestions before displaying them to the user so that we only display those that are available.

Users interact with their previous URLs on the home pageUsers may also modify previously created long and short URL associations. Our implementation of this feature offers flexibility in deciding to update the associations individually. Further, the users can delete entire associations completely.

For additional work, we would like to do the following:

- On the home page, include an image from each webpage to display next to the URL associations, for additional context
- Provide a way for user's to share URLs with others
- Allow users to categorize their links

Part 3: Mobile Responsive Design
The responsive design automatically adjusts the page format and styles to be optimized for different screen sizes. This allows us to serve the same page to a variety of devices such as tablets and smart phones. From the user's perspective, they will look at the site on their various devices see that the page sections stack on top of each other in the mobile view.

To implement this feature, we wrote two css documents. One style sheet has all of the styles for the full desktop version and the other style sheet has the media queries for the tablet and mobile views. With the media queries, we override some of the basic styles for font sizes, div widths, and table styling. The main break points are at 768px for the iPad portrait view and 480px for the iPhone landscape view.

The most difficult part of implementing the responsive design was adding the responsive css after all of the html pages had been designed for desktop. This required redesigning the divs and section widths for all of the pages. In the future, we would recommend planning out the desktop, tablet, and mobile layouts for each page before writing any html or css.

Before we decided to write the css media queries from scratch, we considered using Twitter Bootstrap to design the responsive layouts. It seemed that Twitter Bootstrap has far more functionality that we needed for the small number of pages that we have. In addition, it would take some effort to overriding the Bootstrap styles to really customize our page. All in all, we decided that it would be a good exercise to learn how to create a custom responsive design for our project.

Future improvements would include smoothing out the font-size transitions between the different views. Right now, the font sizes jump and and down when adjusting a browser's screen width over a break point. We would also like to restyle how the table looks in the mobile view, so that the long url and short url stack vertically, rather than just hiding the long url from view.

Part 4: Recommended Short URLs based on Selected Webpage
When generating short URLs, users might be tired of either 1) generating their own short url string that might often have duplicates and get rejected 2) wanting a more semantically useful short URL versus a random set of letters.

The recommended Short URL generator will output a suggested set of usable Short URL keywords when a user enters a long URL and clicks a button activating the recommendation.

There were three main parts to implement the feature:
1) Prefetching Long URL Header Title Information: We determined that the most consistent set of webpage meta-data is contained in the <title> tag in the html header section. When a user enters a long URL, the application sends a http request to extract the header information of the web page. We determined that this was most consistent meta-data information common to most websites. Sometimes there were <meta> information and sometimes not.
2) Extract Common Words from Title: The application filters out very common english words like pronouns from the webpage title
3) Generate Keyword Recommendations Using Twitter Hashtag Frequency: The application is using the frequency of Twitter hashtags (#hashtags) coinciding with Twitter tweets retrieved from Twitte search results from the html title keywords. For example, if a html title contains a proper noun like “Berkeley” when you do a Twitter Search for “Berkeley” you will have a high chance to get hashtags of “#Berkeley”. We generate a histogram and return the most frequent hashtags.

User Case
Twitter searches are very good with article webpages with well developed html titles and refer to recent events.
[Example: Current News Events]
WebPage Title: “<title>North Korea Says Uncle of Kim Jong Un Executed - WSJ.com</title>
Twitter Suggestions: Generates the recommendation “JangSongThaek” which is the actual name of the person executed. This information is not present in the title.
[Example: Popular Trends]
Web Page Title: <title> One Best Actor, So Many Possibilities - NYTimes.com <title>
Twitter Suggestions: Returns “movies”, “DallasBuyersClub” etc. because these are movies related to the movie nomination for best actor.

Current Issues and Future Improvements
We are using basic Twitter API to conduct manual twitter searches. As a result, the application is pretty slow (1 sec to 5 sec) to generate the recommended short URLs. This is because we are sending a large number of AJAX calls to the Twitter Search API and waiting for their resolution to generate the final frequency histogram. To improve

performance, it is possible to create to manually cache the universe of Twitter Tweets for a given period. This allows performance improvement because the application can reduce the latency of the AJAX API calls by conducting the queries using local resources. For additional work, we would also like to show some message to the user to let them know that the results are being generated so they know why it appears as if nothing is happening. We were unable to generate notifications for AJAX determinations due to the many number of chained AJAX calls. In the future, we would like to provide dynamic prompts to let the user know that the application is still waiting for the Twitter results or whether it fails.

Also, the Twitter API has a tendency to throttle search API requests. So, if the site is non-responsive, it might be because Twitter is rejecting the Search API requests.

Also, Twitter's search results are non-deterministic depending on the dynamic nature of recent tweets. Therefore, a request to get url recommendations can generate very different results.

Dependencies:
We used a Twitter library to help with the authentication with the Twitter API. https://github.com/themattharris/tmhOAuth . This library was created by Tim Harris.

Appendix: File Dependencies
Presentation files are available here: URLShorty presentation

For production implementation, we assume that access to these files will be protected by the web-server to prevent malicious intrusion.

All Files are from the relative "/server" path

- Flask: app_combine.py
- Database Interface Function: db_functions.py
- HTML: All files in "/server/templates"
- CSS: All files in "/server/static/css"
- Images: All files in "/server/static/img"
- Javascript: All files in "/server/static/js"