Data Science Internship Feb 2023 Batch

Final Project: URL Shortener Application (Basic)

Name: Shubham Yadav

Introduction:

This URL Shortener App is a web application that allows users to shorten long URLs into shorter, more manageable links. The app was designed to meet specific requirements, which included allowing users to enter a URL, shortening the URL and displaying it in a text-field, saving the URL in a database, and verifying the entered URL's validity. In this report, we will discuss how I implemented each requirement, the challenges faced.

Features:

The app has the following features:

- Shorten URLs using is.gd or bit.ly API.
- Save the URLs to an SQLite database.
- Display list of previously shortened URLs.

Technologies:

The app was developed using the following technologies:

- Flask: A Python web framework for building web apps
- SQL Alchemy: an ORM (Object-Relational Mapping) library for Python
- Flask-Migrate: A Flask extension for managing database migrations
- HTML and CSS
- Bootstrap: A popular CSS framework for building responsive web pages

Implementation:

1. URL Shortening:
    - When a user enters a long URL into the app, they have the option to choose between two URL shortening services: is.gd or Bit.ly. If the user selects is.gd, the app makes an API call to the is.gd service to generate a shortened URL. If the user selects Bit.ly, the app uses the Bit.ly API to generate a shortened URL. The resulting shortened URL is then displayed to the user along with a copy button.

    - The user can manually select and copy the shortened URL or click the copy button on the right-hand side.
2. URL History Tracking:
    - Upon clicking either of the shorten button, the URL entered is saved into our database along with its shortened version.
    - This History can be accessed by navigating to the “History” page in the Navigation of the app.
3. URL verification:
    - To verify the URL, request is used to pass the URL to either of the API. If the URL is valid the API returns with 2xx status code along with the shortened form of the URL passed.
    - If we get a status code other than 2xx we can assume that the URL is invalid or the API is unavailable for some time, In this case this app will show an Error stating “Could not shorten the URL”.
4. The app has 3 main routes / end points:
    - ‘/’ - This route serves as the home page for the app. It allows users to enter a long URL and select a URL shortening service. When the user submits the form, the app generates a shortened URL and displays it to the user. The original and shortened URLs are also added to the database.
    - ‘/history’ - This route displays a list of all URLs that have been shortened by the app, along with their corresponding original URLs and shortened URLs.
    - ‘/clear’ – This route clears all the URLs from the database.

Detailed Steps for Entire Process:

Below are the detailed steps I took along with the challenges I Faced throughout the building of the entire app. I started out with the things I was familiar with and decided to leave the database implementation for the end.

1. Started by creating a Flask project structure for app.py :
    - Step1: Import flask and required modules.
    - Step2: Create object for the Flask class.
    - Step3: Create end-points / routes.
    - Step4: run the app using `app.run(debug=True) `.
    - Also created the templates directory for the html pages.
2. Then Created the layout.html page which will be inherited by all the other pages:
    - Layout.html consists of 2 blocks one for title and one for the content/body and a navbar to traverse between the pages.

- Used bootstrap to make it easier to implement the CSS also the theme for the app is dark.

3. Created home.html - This is where the user will enter the URL and receive a shortened form of the URL:
    - The home.html extends the previously created layout and changes just the contents of the blocks. The title block contains the title appropriate for this page i.e., “Home”.
    - The body consists of a Form element which consists of a Text Input and 2 buttons(for user to choose the service of choice either is.gd or bit.ly).
    - Other than this the home page also consists of a Error Conditions which is triggered if the API responds with a non 2xx status code, there is also a if condition to display the output box and copy button on successful execution of the shorten function.
    - And finally, it also contains a JavaScript function which is an event Handler for the copy button.
4. Created the history.html page - This is where the previously shortened URL will be displayed along with their long versions which are stored in the database:
    - This history.html also extends the previously created layout and changes the title and body block similar to home.html.
    - It consists of a header history and a table with 3 columns viz. the serial No. of the URL entry, the original URL and the shortened URL, the table is populated using a for loop that iterates through the list(in this case the query().all() result) of URLs passed from the backend.
    - If there are No URLs in the database then the webpage displays the message “No URLs found.”
5. Completed the app.py:
    - Imported and Installed the rest of the modules required for database integration.
    - Configured SQLite database using SQLAlchemy, using ->
        i. base = os.path.abspath(os.path.dirname(__file__))
        ii. flask.config[‘SQLALCHEMY_DATABASE_URI’] ='sqlite:///' + os.path.join(base, 'data.sqlite')
        iii. flask.config['SQLALCHEMY_TRACK_MODIFICATIONS']=False
    - Created the SQLAlchemy object and initialize it with the Flask object, then migrate our app to the sqlite database->
        i. db = SQLAlchemy(flask)
        ii. Migrate(flask,db)
    - Created Model for the URLs using SQLAlchemy.

    i. For this we create a class for the Model which defined the Schema of the sqlite table.
    ii. We also define a constructor for the class and a string representation for the object.
- Defined a route for homepage.
    i. Here, if the method of the request is “GET” the app renders home.html to the Frontend which consists of a Text Input box along with 2 Buttons for 2 service APIs.
    ii. And if the request is “POST”(which happens when the user submits a URL to the input and clicks on either of the button), the backend uses request to get the submitted URL and service based on the button clicked.
    iii. After clicking either of the shorten button the app also checks if the URL starts with “http” if it does not then the URL is modified and “https://” is added to the start.
    iv. Based on the service picked the appropriate function is called to request the corresponding API.
- Defined the function to shorten URLs using is.gd API.
    i. This function uses is.gd API to request a shortened form of the URL user provided.
    ii. If the status code returned by the API is 2xx then response.text will contain our shortened link which will be returned back to the short_url variable, and if the status code is anything other than 2xx then the function reurns None type.
- Defined the function to shorten URLs using bit.ly API.
    i. Similar to is.gd function but using the API and Access token for bit.ly.
- Defined the route for the History page and query all the URLs in the database.
    i. This route queries the entire sqlite database for Urls and renders history.html along with the result of the query.
    ii. To query the entire table Url.query.all() is used (Equivalent to SELECT * FROM Table_Name).
- Also defined the route to clear all the entries in the database.
    i. This route deletes all the entries from the table and returns an empty History page.
- Run the app.
    i. The final step of any app.

That's it! This code defines a Flask app that allows users to shorten URLs using either the is.gd or Bitly API and stores the URLs in a SQLite database. It also allows users to view the history of shortened URLs and clear all the entries in the database.