URL Shortener Web Application (Basic)

Step-1: Setting up dependencies

1. Creating environment using command

   python -m venv env

2. Activate the environment

   env\Scripts\activate

Step-2: Installing Flask and hashid

pip install flask

pip install hashids

Step-3: Creating a database schema file called schema.sql, containing SQL commands to create a urls table

In this file we drop table urls if it exists already

- id: The ID of the URL to be unique is set as primary key ,We will use it to get the original URL.
- created: The date when shortened URL is created
- original_url: The original long URL
- clicks: The number of times a URL has been clicked. The initial value will be 0, which will increment with each redirect.

To execute the schema.sql file to create the urls table, we open a file named init_db.py

Here you connect to a file called database.db that your program will create once you execute this program. This file is the database that will hold all of your application's data. schema.sql file is run using the executescript() method that executes multiple SQL statements at once. This will create the urls table. Finally, we commit the changes and close the connection

Step-4: Creating database.db

```
python init_db.py
```

Step-5: Creating index page

app.py is created under directory named URL-SHORTENER

We first import the sqlite3 module, the Hashids class from the hashids library, and Flask helpers.

The get_db_connection() function opens a connection to the database.db database file and then sets the row_factory attribute to sqlite3.Row. As a result, we can have name-based access to columns; the database connection will return rows that behave like regular Python dictionaries. Lastly, the function returns the conn connection object we'll be using to access the database

We create the Flask application object and set a secret key to secure sessions. Since the secret key is a secret random string, we'll also use it to specify a *salt* for the Hashids library; this

will ensure the hashes are unpredictable since every time the salt changes, the hashes also change.

A salt is a random string that is provided to the hashing function (that is, hashids.encode()) so that the resulting hash is shuffled based on the salt. This process ensures the hash we get is specific to your salt so that the hash is unique and unpredictable, like a secret password that only we can use to encode and decode hashes.

We create a hashids object specifying that a hash should be at least 4 characters long by passing a value to the min_length parameter. We use the application's secret key as a salt.

The index() functions is a Flask *view function*, which is a function decorated using the special @app.route decorator. Its return value gets converted into an HTTP response that an HTTP client, such as a web browser, displays.

Inside the index() view function, we accept both GET and POST requests by passing methods=('GET', 'POST') to the app.route() decorator. We open a database connection.

Then if the request is a GET request, it skips the if request.method == 'POST' condition until the last line. This is where we render a template called index.html, which will contain a form for users to enter a URL to shorten.

If the request is a POST request, the if request.method == 'POST' condition is true, which means a user has submitted a

URL. we store the URL in the url variable; if the user has submitted an empty form, we flash the message The URL is required! and redirect to the index page.

If the user has submitted a URL, we use the INSERT INTO SQL statement to store the submitted URL in the urls table. We include the ? placeholder in the execute() method and pass a tuple containing the submitted URL to insert data safely into the database. Then we commit the transaction and close the connection.

In a variable called url_id, we store the ID of the URL we inserted into the database. We can access the ID of the URL using the lastrowid attribute, which provides the row ID of the last inserted row.

We construct a hash using the hashids.encode() method, passing it the URL ID; you save the result in a variable called hashid. We then construct the short URL using request.host_url, which is an attribute that Flask's request object provides to access the URL of the application's host. This will be http://127.0.0.1:5000/ in a development environment and your_domain if We deploy our application

Step-6: Creating base.html and index.html

base.html contains the code which is extended in index and about.html using jinja template. And it also contains navigation bar where HOME and ABOUT redirects to index.html and about.html pages (rendering with respect to their functions)

Step-7: Redirecting route

new route will also use the integer ID to fetch the original URL and increment the clicks value. Finally,we will redirect users to the original URL.

This new route accepts a value id through the URL and passes it to the url_redirect() view function. For example, visiting http://127.0.0.1:5000/io90 would pass the string 'io90' to the id parameter.

Inside the view function, we first open a database connection. Then we use the decode() method of the hashids object to convert the hash to its original integer value and store it in the original_id variable. we check that the original_id has a value—meaning decoding the hash was successful. If it has a value, we extract the ID from it. As the decode() method returns a tuple, we fetch the first value in the tuple with original_id[0], which is the original ID.

we then use the SELECT SQL statement to fetch the original URL and its number of clicks from the urls table, where the ID of the URL matches the original ID we extracted from the hash. We fetch the URL data with the fetchone() method. Next, we extract the data into the two original_url and clicks variables.

we then increment the number of clicks of the URL with the UPDATE SQL statement.

we commit the transaction and close the connection, and redirect to the original URL using the redirect() Flask helper function.

If decoding the hash fails, we flash a message to inform the user that the URL is invalid, and redirect them to the index page.

Step-8: Creating stats page

we'll add a new route for a statistics page that displays how many times each URL has been clicked. we'll also add a button that links to the page on the navigation bar.

Allowing users to see the number of visits each shortened link has received will provide visibility into each URL's popularity

We open a database connection. Then we fetch the ID, the creation date, the original URL, and the number of clicks for all of the entries in the urls table. we use the fetchall() method

to get a list of all the rows. we then save this data in the db_urls variable and close the connection.

To display the short URL for each entry,we will need to construct it and add it to each item in the list of the URLs we fetched from the database (db_urls). We create an empty list called urls and loop through the db_urls list with for url in db_urls.

we use the dict() Python function to convert the sqlite3.Row object to a dictionary to allow assignment. we add a new key called short_url to the dictionary with the value request.host_url + hashids.encode(url['id']), which is what we used before to construct short URLs in the index view function. we append this dictionary to the urls list.

Finally, we render a template file called stats.html, passing the urls list to it.

defining a table with the following columns:

- S.NO: The ID of the URL.
- Short-URL: The short URL.
- Original_URL: The original URL.
- Clicks: The number of times a short URL has been visited.
- Creation Date: The creation date of the short URL.

Step-9: Running the App

Python -m flask run