URL Shortener Using Flask Framework and SQlite3

We use Flask, SQLite, and the Hashids library to build your URL shortener. Your application will allow users to enter a URL and generate a shorter version, in addition to a statistics page where users can view the number of times a URL has been clicked. You'll use the Bootstrap toolkit to style your application

Step 1:

Then create a database schema file called `schema.sql`, containing SQL commands to create a `urls` table. Open a file called `schema.sql` inside your `urlshortner` directory:

In the schema file, you first delete the `urls` table if it already exists. This avoids the possibility of another table named `urls` existing, which might result in confusing behavior; for example, if it has different columns. Note that this will delete all of the existing data whenever the schema file executes.

You then create the table with the following columns:

- `id`: The ID of the URL, this will be a unique integer value for each URL entry. You will use it to get the original URL from a hash string.
- `created`: The date the URL was shortened.
- `original_url`: The original long URL to which you will redirect users.
- `clicks`: The number of times a URL has been clicked. The initial value will be `0`, which will increment with each redirect.

Save and close the file.

To execute the `schema.sql` file to create the `urls` table, open a file named `init_db.py` inside your `flask_shortener` directory

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. You then open the `schema.sql` file and run it using the `executescript()` method that executes multiple SQL statements at once. This will create the `urls` table. Finally, you commit the changes and close the connection.

Save and close the file.

Step 2:

In this step, you will create a Flask route for the index page, which will allow users to enter a URL that you then save into the database. Your route will use the ID of the URL to generate a short string hash with the Hashids library, construct the short URL, and then render it as a result.

First, open a file named `app.py` inside your `flask_shortener` directory. This is the main Flask application file:

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, you accept both GET and POST requests by passing `methods=('GET', 'POST')` to the `app.route()` decorator. You 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 you 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. You store the URL in the `url` variable; if the user has submitted an empty form, you flash the message `The URL is required!` and redirect to the index page.

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

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

You construct a hash using the `hashids.encode()` method, passing it the URL ID; you save the result in a variable called `hashid`. As an example, the call `hashids.encode(1)` might result in a unique hash like `KJ34` depending on the salt you use.

You 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 you deploy your application. For example, the `short_url` variable will have a value like `http://127.0.0.1:5000/KJ34`, which is the short URL that will redirect your users to the original URL stored in the database with the ID that matches the hash `KJ34`. Lastly, you render the `index.html` template passing the `short_url` variable to it

Step 3:

In this step, you will add a new route that takes the short hash the application generates and decodes the hash into its integer value, which is the original URL's ID. Your new route will also use the integer ID to fetch the original URL and increment the `clicks` value. Finally, you 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/KJ34` would pass the string `'KJ34'` to the `id` parameter.

Inside the view function, you first open a database connection. Then you 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. You check that the `original_id` has a value—meaning decoding the hash was successful. If it has a value, you extract the ID from it. As the `decode()` method returns a tuple, you fetch the first value in the tuple with `original_id[0]`, which is the original ID.

You 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 you extracted from the hash. You fetch the URL data with the `fetchone()` method. Next, you extract the data into the two `original_url` and `clicks` variables.

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

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

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

Save and close the file.

Run your development server:

Step 4:

In this step, you'll add a new route for a statistics page that displays how many times each URL has been clicked. You'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, which is useful for projects, like marketing ad campaigns. You can also use this workflow as an example of adding a feature to an existing Flask application.

Open `app.py` to add a new route for a statistics page:

Conclusion

You have created a Flask application that allows users to enter a long URL and generate a shorter version. You have transformed integers into short string hashes, redirected users from one link to another, and set up a page for statistics so you can monitor shortened URLs. For further projects and tutorials on working with Flask, check out the following tutorials: