URL Shortner Application

Final Project Report

AYESHA FARHEEN

INNOMATICS RESEARCH LAB  DS INTERN 23 FEB

URL SHORTENING APP REPORT

Introduction :

URL shortening is the process of shortening long URLs (Uniform Resource Locators) to less complex, easy-to-use URLs. It has become prevalent in recent years because it is an easy way to share URLs, particularly on social media platforms, where character counts can be very limited. A URL shortener will help in shortening the URL but it also allows the user to copy the shortened URL with a click of a button. The purpose of this report is to outline the development process of a web application that can shorten URLs.

Problem Statement:

The primary goal of this project is to develop a web app that can shorten URLs. The web app is divided into three parts: Frontend, Backend, and ORM. The frontend will be done with HTML, CSS, and Bootstrap, the backend will be done with Flask (Python), and the ORM will be done with SQLAlchemy. The database for this web app will be SQLite.

Objectives :

The objectives of this web app are as follows:

Shorten URLs: The main objective of this web app is to shorten URLs. Users can enter long URLs, and the web app will shorten them into more manageable URLs.

Save URLs: Users can save URLs by logging into the web app. After logging in, the user can enter the URL they want to shorten, and the shortened URL will be displayed. The web app will save the entered URL and its corresponding shortened URL in the database, which the user can later access.

User Authentication: The web app will have user authentication, where users can create an account or log in if they already have an account. The user will be able to log in and save their URLs.

Features:

The features of this web app are as follows:

-Sign-up page

The user can create an account by entering a unique username. The length of the username is valid only in the range of 5 and 9. If the username is not of valid length, then an error message will occur: “Username must be between 5 to 9 characters long”.

-Log-in page:

After creating an account, the user can log in with their username and password.

-Shorten URL:

After logging in, the main web-application opens which is the URL shortener. Here the user can enter the URL they want to shorten. After entering a URL, click on the ‘shorten’ URL button to display the shortened URL in the following text-field which can be copied by clicking on the copy button.

-Copy shortened URL:

After the ‘shorten’ button is clicked, the shortened URL is displayed in the text-field which the user can copy using the copy button.

-Save URL:

After the ‘shorten’ button is clicked, the URL that is entered is saved in our database with the shortened URL. It is saved in the database so that the user can look into the previous URLs they entered in our web-app with their shortened URL in the forms given below the copy button.

Implementation:

The implementation of the URL Shortener Web App is divided into three parts: Frontend, Backend, and ORM.

-Frontend :

The frontend of this web app is created with HTML, CSS, and Bootstrap. The web app consists of three webpages: Home, Signup, and Login.

-Home:

This is the first page that the user will see. If the user already has an account, they can log in. Otherwise, they can create an account by clicking on the Sign-up button.

-Signup:

This page is used for creating an account. The user needs to enter a unique useremail. If the user enters a useremail that is already taken, then an error message will occur displaying “This user already exists…” Then the user needs.

The approach to make a url shortener application can be divided into several steps:

1.Setting up the Flask application:

The first step would be to set up the Flask application by importing the necessary libraries, configuring the database, and creating the required models.

2.Creating the Front-end:

The next step would be to create the front-end of the web application using HTML, CSS, and Bootstrap. This would involve creating the login and signup pages, as well as the main URL shortening page.

3.Creating the Backend:

After the front-end is created, the next step would be to create the backend of the web application using Flask. This would involve creating routes for the different pages and implementing the necessary functionality for user authentication and URL shortening.

4.Integrating with SQLAlchemy:

The next step would be to integrate the Flask application with SQLAlchemy to provide ORM functionality for the database.

5.Implementing User Authentication:

The next step would be to implement user authentication, which would involve creating a User model and implementing functionality for user registration, login, and logout.

6.Implementing URL Shortening:

The next step would be to implement the functionality for URL shortening, which would involve creating a Url model and implementing functionality for generating and storing the shortened URLs in the database.

7.Testing and Debugging:

Finally, the web application would need to be thoroughly tested and debugged to ensure that it works correctly and is free of any errors or bugs.

These are the steps to follow in order to implement url shortener application:

1. Importing necessary libraries:

import os

from flask import Flask, render\_template, request, redirect, url\_for, flash

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

from flask\_login import LoginManager, UserMixin, login\_required, login\_user, logout\_user, current\_user

from werkzeug.security import generate\_password\_hash, check\_password\_hash

import pyshorteners

* os is a Python module used for interacting with the operating system.
* Flask is a web framework for Python.
* render\_template is used for rendering HTML templates.
* request is used for handling HTTP requests.
* redirect is used for redirecting to another URL.
* url\_for is used for generating URLs for endpoints.
* flash is used for displaying messages to the user.
* SQLAlchemy is an Object-Relational Mapping (ORM) library for Python.
* Migrate is a library used for handling database migrations with Flask and SQLAlchemy.
* LoginManager is a Flask extension that provides user session management.
* UserMixin is a class provided by Flask-Login that adds default implementations for the methods required by the user model.
* login\_required is a decorator that can be used to require authentication before accessing a view function.
* login\_user is a function that can be used to log in a user.
* logout\_user is a function that can be used to log out a user.
* current\_user is a variable that holds the current user object.
* generate\_password\_hash and check\_password\_hash are functions used for password hashing.
* pyshorteners is a library used for shortening URLs.

2.Creating an instance of the Flask class:

app = Flask(\_\_name\_\_)

This creates a Flask application instance.

3.SQLAlchemy configuration:

basedir = os.path.abspath(os.path.dirname(\_\_file\_\_))

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///' + os.path.join(basedir, 'data.sqlite')

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

app.config['SECRET\_KEY'] = 'mykey'

db = SQLAlchemy(app)

Migrate(app, db)

* basedir is set to the absolute path of the directory containing the current file.
* SQLALCHEMY\_DATABASE\_URI is set to a SQLite database file located in the basedir.
* SQLALCHEMY\_TRACK\_MODIFICATIONS is set to False to disable modification tracking.
* SECRET\_KEY is set to a string used for securely signing cookies.
* An instance of SQLAlchemy is created and passed the app instance.
* An instance of Migrate is created and passed the app and db instances.

4.Login manager configuration:

login\_manager = LoginManager()

login\_manager.init\_app(app)

# Tell users what view to go to when they need to login.

login\_manager.login\_view = "login"

* An instance of LoginManager is created and passed the app instance.
* login\_view is set to the name of the view function that handles login requests.

5.Creating a User model:

class User(db.Model, UserMixin):

    id = db.Column(db.Integer, primary\_key=True)

    email = db.Column(db.String(120), unique=True, nullable=False)

    password\_hash = db.Column(db.String(128), nullable=False)

    @property

    def is\_active(self):

        return True

    def check\_password(self, password):

        return check\_password\_hash(self.password\_hash, password)

* This creates a User model that inherits from db.Model and UserMixin.

Following are the routes that are being created in the app.py file of Flask.

@app.route('/')

- This route is associated with the welcome() function, which renders the welcome.html template when the user visits the homepage.

@app.route('/signup', methods=['GET', 'POST'])

- This route is associated with the signup() function, which handles user sign-up requests. If the request is a GET request, the signup.html template is rendered. If the request is a POST request, the function creates a new user account and adds it to the database, and then redirects the user to the login page.

@app.route('/login', methods=['GET', 'POST'])

- This route is associated with the login() function, which handles user login requests. If the request is a GET request, the login.html template is rendered. If the request is a POST request, the function checks whether the email and password are valid and matches with the user's credentials. If it matches, the user is logged in and redirected to the welcome page. If it doesn't match, the user is redirected back to the login page with an error message.

@app.route('/shorten', methods=['GET', 'POST'])

- This route is associated with the shorten() function, which handles URL shortening requests. If the request is a GET request, the Application.html template is rendered with an empty short\_url variable. If the request is a POST request, the function shortens the URL using the pyshorteners library and adds the original and shortened URL to the database. The short\_url variable is updated with the shortened URL, and then the Application.html template is rendered with the updated short\_url variable.

@app.route('/history', methods=['GET', 'POST'])

- This route is associated with the history() function, which handles URL history requests. If the request is a GET request, the function retrieves all URL records from the database associated with the current user and passes them to the history.html template. If the request is a POST request, the function handles the form submission, although in the current implementation, there is no code for handling form submissions.

@app.route('/logout')

– This route is associated with the logout() function, which logs the current user out and redirects them to the welcome page.

THANK YOU