URL Shortener Web Application (Advanced Version)

1. Introduction

This project focuses on building an Advanced URL Shortener Web Application using the Flask framework in Python. Unlike a basic URL shortener, this advanced version introduces user authentication and session management, allowing users to register, log in, log out, and manage their own shortened URLs on a per-user access control.

In real-world applications, URL shorteners are widely used to convert long and complex URLs into short, shareable links. However, without authentication, it becomes difficult to manage URLs on a per-user basis. This advanced implementation solves that limitation by associating shortened URLs with individual user accounts, making the system more structured, secure, and closer to real-world web applications.

This project was developed as part of the Flask backend development module under the Innomatics internship program. The primary goal was to gain hands-on experience with backend development concepts such as authentication, protected routes, session handling, database integration, and user-specific data management using Flask.

2. Project Objectives

The main objectives of this advanced project are:

- To design and implement an advanced URL shortener using Flask.
- To allow users to register (signup) with a username and password.
- To enable secure login and logout functionality.
- To restrict URL shortening and history access to authenticated users using Flask-Login protected routes.
- To store original URLs & shortened URLs in the database and associate them with the logged-in user.
- To display a user-specific history showing only the logged-in user's URLs.
- To redirect users to the original URL when a shortened URL is accessed.

These objectives ensure that the application is secure, user-specific, and aligned with real-world backend application requirements.

3. Technologies Used

The following technologies and tools were used in this project:

- Python : Used as the core programming language for backend logic.
- Flask : A lightweight web framework used for routing, request handling, sessions, and responses.
- Flask-Login : Used to manage user authentication, login sessions, protected routes, and logout functionality.
- HTML : Used to design the frontend pages such as login, signup, home, and history.
- Jinja2 : Flask's templating engine used to dynamically render data on HTML pages.
- SQLite : A lightweight database used to store user credentials and URL mappings.
- Werkzeug : Used for password hashing and secure credential handling.

These technologies were chosen because they are beginner-friendly, lightweight, and suitable for small to medium-scale backend applications.

4. Application Architecture

The follows a clean & modular structure, ensuring separation of concerns & easy debugging.

- **app.py**
    - Flask app initialization
    - User data handling (authentication and credential storage)
    - URL data handling (original and shortened URL storage)
    - Authentication routes (Signup, Login, Logout)
    - URL shortening logic
    - User-specific history retrieval
    - URL redirection logic
- **templates folder (**Contains HTML files**)**
    - login.html – User login page
    - signup.html – User registration page
    - index.html – Home page for URL shortening (accessible only after login)
    - history.html – Page showing user-specific URL history
- **instance folder**
    - Contains the SQLite database file, where user details and URL mappings are stored securely.

This architecture improves readability, scalability, and overall maintainability of the application.

5. Database Design

The application uses SQLite as the database because it is lightweight, easy to configure, and suitable for small-scale applications.

• Original URLs • Shortened URLs
• User credentials (username and hashed password)
• The user identifier associated with each shortened URL

Each shortened URL is linked to the user who created it using a stored user identifier, which allows the application to display URL history specific to the logged-in user.

6. URL Validation and Redirection Logic

Before storing any URL:

- The application performs basic validation to ensure the URL input is not empty before processing.
- Only valid inputs are stored in the database.

For redirection:

- Flask captures the short code from the URL.
- The database is searched for a matching record.
- If a matching URL is found, the user is redirected to the original URL.
- If no match is found, the request is handled safely without crashing the application.

This logic ensures correct URL mapping and reliable redirection behavior.

7. Application Workflow (Step-by-Step)

User Registration (Signup)

- The user opens the signup page.
- The user enters a username and password.
- The application checks:
  - Whether the username already exists.
  - Whether the username length is within the allowed range.
- If validation passes, the password is securely hashed.
- The new user details are stored in the database.
- The user is redirected to the login page.

User Login

- The user enters username and password on the login page.
- The application verifies the credentials.
- If valid, a login session is created using Flask-Login.
- The user is redirected to the home page.
- Unauthorized users are prevented from accessing protected routes.

URL Shortening

- The logged-in user enters a long URL on the home page.
- The application validates the input.
- A unique short code is generated.
- The shortened URL is created using the base URL and short code.
- The original URL, shortened URL, and user ID are stored in the database.
- The shortened URL is displayed on the page for copying.

URL History

- The user clicks on “View History”.
- The application fetches only the URLs created by the logged-in user.
- The history page displays:

  Original URLs
  Corresponding shortened URLs

- Clicking a shortened URL redirects to the original URL.

User Logout

- The user clicks the logout option.
- The login session is cleared.
- The user is redirected to the login page.
- Protected pages are no longer accessible without logging in again.

8. Challenges Faced and Debugging

During development, several challenges were encountered:

- Understanding and implementing Flask-Login correctly.
- Managing protected routes and login-required access.
- Handling user sessions across multiple pages.
- Ensuring username uniqueness during signup.
- Resolving dependency issues related to Python and pip.
- Connecting authentication logic with URL storage.
- Displaying user-specific history instead of global data.

All challenges were resolved through documentation study, debugging, testing, and hands-on practice.

9. Learning Outcomes

Through this project, the following learning outcomes were achieved:

- Gained strong practical knowledge of Flask backend development.
- Learned how authentication systems work in web applications.
- Understood session handling and protected routes.
- Learned how to securely store user credentials.
- Improved database design and relational thinking.
- Gained confidence in building complete backend-driven applications.

10. Conclusion

This project helped in understanding how an advanced web application works from start to end using Flask.

By working on this Advanced URL Shortener application, practical knowledge was gained about user authentication, session management, handling protected routes, and managing user-specific data securely within a web application.

Building this application also improved confidence in backend development, especially in implementing login and signup functionality, integrating Flask-Login, working with HTML templates, and connecting user data with database records using SQLite.

The project provided hands-on experience in solving real-world problems such as securing access to application features, managing URLs on a per-user basis, and ensuring proper redirection and session handling.

Overall, this project was a valuable learning experience and created a strong foundation for developing secure and scalable web applications in the future.