Project Report: URL Shortener (Advanced)

Submitted by: Arshad Murtaza

1. Objective The goal of this project was to build a URL Shortener application with Authentication capabilities. The app allows users to Sign Up, Log In, and manage their own private list of shortened URLs.

2. Architecture & Database I used a "One-to-Many" database relationship using SQLAlchemy.

- **User Table:** Stores `id`, `username`, and `password`.
- **Url Table:** Stores `id`, `original_url`, `short_url`, and a `user_id`.
- **Relationship:** The `user_id` in the Url table is a Foreign Key linking back to the User table. This ensures that when a user logs in, we can query `Url.query.filter_by(user_id=current_user.id)` to show only *their* links.

3. Key Features & Logic

- **Authentication:** I used `Flask-Login` to manage user sessions. This restricts the main dashboard so that only logged-in users can access it (`@login_required`).
- **Username Validation:** As per the requirements, I added logic in the Signup route to check:
    1. If the username length is between 5 and 9 characters.
    2. If the username is already taken in the database.
- **URL Shortening:** The app generates a random 5-character string. Before saving, it checks the database to ensure this random string hasn't been used before.

4. Challenges Solved

- **Constraint Handling:** The requirement to limit usernames to 5-9 characters was handled using a simple Python `len()` check before creating the database entry.
- **User Separation:** Initially, all URLs were visible to everyone. I solved this by adding the `owner` relationship in the database models, ensuring data privacy.

4. **Conclusion** The application successfully meets all "Advanced User" requirements, providing a secure and personalized experience for shortening URLs.

PROJECT REPORT

Advanced URL Shortener Web Application

Submitted By: Arshad Murtaza **Date:** 21-01-2026

---

1. Abstract

The "Advanced URL Shortener" is a web-based application designed to convert long, cumbersome Uniform Resource Locators (URLs) into concise, manageable links. Unlike basic utility tools, this application incorporates a secure user authentication system, ensuring that link management is personalized and private. The system was developed using Python's Flask framework for the backend and SQLite for data management, adhering to modern Model-View-Controller (MVC) architectural patterns. This report documents the development lifecycle, from requirement analysis and database design to implementation and testing.

2. Introduction

2.1. Problem Statement In digital communication, sharing deep-links to specific web pages often results in extremely long strings of characters. These links are aesthetically unpleasing, prone to breaking when copied across lines in emails or SMS, and difficult to remember. Furthermore, without a user account system, users cannot track the links they have generated in the past or manage their data effectively.

2.2. Proposed Solution The developed solution is a full-stack web application that addresses these issues by:

- Generating unique, 5-character alphanumeric codes for any given URL.
- Providing a **User Authentication System** (Signup/Login) to secure user data.
- Implementing strict validation rules (e.g., username length constraints) to ensure data integrity.
- maintaining a history log where users can view their previously shortened links.

3. System Analysis & Requirements

3.1. Functional Requirements

- **User Registration:** New users must be able to sign up. The system must enforce a constraint that usernames are between 5 and 9 characters long.
- **Authentication:** Users must log in to access the shortening features. Access to the dashboard must be restricted to authenticated users only.
- **URL Shortening:** The system must accept a valid URL and return a unique shortened version.
- **Redirection:** Entering the short code in the browser must redirect the user to the original destination.

- **History Management:** Users must be able to see a list of all URLs they have created.

3.2. Technology Stack

- **Language:** Python 3.x (Chosen for its simplicity and robust library support).
- **Framework:** Flask (A micro-framework ideal for rapid development of lightweight web apps).
- **Database:** SQLite (Serverless database, integrated via SQLAlchemy ORM).
- **Frontend:** HTML5, CSS3, and Bootstrap 5 (For responsive design).

[PAGE 2: SYSTEM DESIGN & IMPLEMENTATION]

4. System Design

4.1. Database Schema The application uses a Relational Database Management System (RDBMS). We utilized **SQLAlchemy ORM** to define two primary models with a "One-to-Many" relationship:

- **Table 1: User**
  - `id` (Integer, Primary Key): Unique identifier for the user.
  - `username` (String): The user's login name (Unique).
  - `password` (String): Stores the user's password.
  - *Relationship:* One User can possess many URLs.
- **Table 2: Url**
  - `id` (Integer, Primary Key): Unique identifier for the record.
  - `original_url` (String): The long link provided by the user.
  - `short_url` (String): The generated unique code (e.g., `abc12`).
  - `user_id` (Integer, Foreign Key): Links the URL to the specific User who created it.

4.2. Application Architecture The application follows the MVT (Model-View-Template) architecture specific to Flask:

1. **Model:** Defines the structure of the database (User and Url classes).
2. **View (Routes):** `app.py` contains the logic. It handles requests, validates input (e.g., checking if a username is unique), and interacts with the database.
3. **Template:** The `templates/` folder contains HTML files (`dashboard.html`, `login.html`) which render the user interface dynamically using Jinja2.

5. Implementation Details

5.1. Authentication Module Security was implemented using `Flask-Login`. The `login_required` decorator is used to protect the main dashboard route. When a user signs up, the system checks two specific constraints requested in the requirements:

1. **Uniqueness:** It queries the database to ensure the username does not already exist.
2. **Length:** It verifies `5 <= len(username) <= 9` before allowing account creation.

5.2. URL Shortening Logic The core functionality relies on the `random` and `string` libraries.

- **Input Validation:** The system checks if the input starts with `http://` or `https://` and appends it if missing to ensure valid redirection.
- **Generation:** A custom function generates a random string of 5 characters.
- **Collision Check:** Before saving, the system queries the database to ensure this 5-character string has not been used before, ensuring 100% uniqueness.

6. Screenshots & Testing

In this section, valid test cases are demonstrated.

Figure 1: The Login Screen *(Insert a screenshot of your Login page here. Crop it nicely so it looks professional.)*

Figure 2: The Dashboard with History *(Insert a screenshot of the Dashboard showing the input box and 2-3 items in the history list.)*

[PAGE 3: CONCLUSION]

7. Challenges Faced

One of the primary challenges was implementing the user-specific history. Initially, all URLs were visible to all users. To resolve this, I implemented a Foreign Key relationship in the database. Now, when the "History" page loads, the query `Url.query.filter_by(user_id=current_user.id)` is executed, ensuring strict data privacy.

8. Conclusion & Future Scope

The "Advanced URL Shortener" project successfully meets all the objectives outlined in the internship problem statement. It provides a secure, user-friendly interface for link management. **Future enhancements could include:**

- **Analytics:** Tracking how many times a short link is clicked.
- **Custom Aliases:** Allowing users to choose their own short words (e.g., `mysite.com/offer`) instead of random characters.
- **Password Hashing:** Improving security by hashing passwords before storage.

This project provided valuable hands-on experience with full-stack development, specifically in connecting a Python backend with a SQL database and managing user sessions.