URL Shortener

1. Project Overview

The objective was to develop a web-based URL shortening service targeting advanced users. The application serves two primary functions: shortening long URLs into compact, shareable links and maintaining a history of these links for registered users. The project was constrained to a specific technology stack: Flask (Backend), SQLAlchemy (ORM), SQLite (Database), and HTML/Bootstrap (Frontend).

2. System Architecture & Tech Stack

The solution was approached using the Model-View-Controller (MVC) architectural pattern, adapted for Flask (which often blends View and Controller).

- **Backend (Controller):** Python/Flask was selected for its lightweight routing capabilities.
- **Database (Model):** SQLite was chosen for its serverless, zero-configuration setup, managed via SQLAlchemy to abstract raw SQL queries into Python objects.
- **Frontend (View):** Jinja2 templating was used to dynamically render HTML, allowing the server to inject user-specific data (like URL history) directly into the webpage.

3. Database Schema Design

The first step in the solution was defining the data models to support the relationship between Users and their URLs. A One-to-Many relationship was implemented.

The User Model

This table handles authentication.

- **Constraint Implementation:** The username field was set to a maximum of 9 characters at the database level to enforce the project's strict length requirement.
- **Primary Key:** An integer id serves as the unique identifier.

The Url Model

This table stores the core business data.

- **Foreign Key:** A user_id column was added to link every URL to a specific creator. This is crucial for the "History" feature.
- **Indexing:** The short_id field was marked as unique=True to prevent two different URLs from receiving the same short code.

4. Implementation Strategy

A. Authentication & Validation Logic

The most complex constraint was the specific validation rule for the signup process: Username must be between 5 and 9 characters.

Approach:

Instead of relying solely on frontend HTML validation (which can be bypassed), I implemented server-side validation in the /signup route.

1. **Length Check:** Before querying the database, the code checks if len(username) < 5 or len(username) > 9. If this fails, the process halts, and a flash message is returned.
2. **Uniqueness Check:** If the length is valid, the database is queried: User.query.filter_by(username=username).first(). If a record returns, the username is taken.
3. **Atomic Commit:** Only if both checks pass is the new user committed to the database.

B. The Shortening Algorithm

To generate the unique short ID, I utilized Python's random and string libraries.

- **Logic:** A function generate_short_id() selects 6 random alphanumeric characters.
- **Collision Handling:** Although statistically rare, it is possible to generate a duplicate ID. I implemented a while loop that checks the database for the generated ID. If it exists, the function runs again until a unique ID is found, ensuring data integrity.

C. User History & Dashboard

To satisfy the requirement that users can see *their* specific URLs, I utilized Flask-Login's current_user proxy.

- **Query Strategy:** instead of loading all URLs (Url.query.all()), the query filters by the logged-in user: Url.query.filter_by(user_id=current_user.id).
- **Rendering:** This list is passed to the Jinja2 template, which iterates through the objects to create the history table seen on the dashboard.

5. Security Considerations

- **Session Management:** Flask-Login was integrated to handle session cookies automatically, ensuring that users remain logged in across pages and cannot access /dashboard without authentication (@login_required).
- **CSRF Protection:** While basic for this scope, the structure allows for future implementation of CSRF tokens in the forms.

6. Conclusion

The resulting application meets all functional requirements: it restricts usernames to the 5-9 character range, securely manages user sessions, and correctly associates shortened URLs with their creators. The modular code structure allows for easy scalability if future features (like click analytics) are required.