Advanced URL Shortener Web Application

Project Report

1. Introduction

With the rapid growth of the internet, sharing digital resources has become a daily activity. URLs are fundamental for accessing online content; however, many URLs are long, complex, and difficult to share. Long URLs can be inconvenient to copy, prone to errors, and visually unappealing, especially when shared via emails, messaging applications, or social media platforms.

A URL Shortener addresses these challenges by converting long URLs into short, unique links that redirect users to the original destination. In addition to shortening URLs, modern URL shorteners also provide features such as user authentication, URL history management, analytics, and security mechanisms.

The objective of this project was to design and implement an **Advanced URL Shortener Web Application** using **Flask**, **SQLAlchemy ORM**, and **Bootstrap**. The project extends a basic URL shortener by introducing authentication, session handling, custom short URLs, click analytics, and database-driven user-specific data storage. This project demonstrates real-world backend development practices and aligns with industry standards.

2. Problem Statement

Although URL shorteners are widely used, implementing one from scratch involves solving multiple backend challenges such as validation, database management, security, and scalability. The problem addressed in this project is to build a URL shortener that not only shortens URLs but also provides an advanced, user-centric experience.

The main objectives of the project are:

1. To shorten long URLs into compact links.
2. To allow users to register and log in securely.
3. To store shortened URLs on a per-user basis.
4. To validate user input and prevent invalid URLs.
5. To support custom short URL creation.
6. To track the number of clicks for each shortened URL.
7. To manage user sessions securely with expiry.
8. To implement clean database interaction using SQLAlchemy ORM.

3. System Architecture

The application follows a **three-tier architecture**, separating concerns for better maintainability and scalability.

3.1 Presentation Layer (Frontend)

The frontend is developed using HTML, CSS, and Bootstrap. Bootstrap was chosen to provide a clean, responsive, and professional user interface with minimal custom styling. The frontend consists of:

- Signup Page
- Login Page
- Dashboard (URL shortening and history)
- Error Page

User input is collected through forms and sent to the backend using HTTP POST requests.

3.2 Application Layer (Backend)

The backend is built using Flask, a lightweight and flexible Python web framework. Flask handles:

- Routing
- Request validation
- Authentication logic
- URL shortening
- Session management
- Redirection logic
- Error handling

The backend logic is modularized into multiple files (app.py, auth.py, utils.py) to improve readability and maintainability.

3.3 Data Layer (Database)

The data layer uses **SQLite** as the database and **SQLAlchemy ORM** for database interaction. ORM abstracts raw SQL queries and allows interaction with database tables using Python classes. This reduces complexity and improves security.

4. Database Design

Two primary database models were designed:

4.1 User Table

Stores user authentication details.

Column	Description
id	Primary key
username	Unique username

Column	Description
password	Hashed password

4.2 URL Table

Stores URL mappings and analytics.

Column	Description
id	Primary key
original_url	Original long URL
short_code	Unique short URL code
clicks	Number of redirects
user_id	Foreign key referencing User

A **one-to-many relationship** exists between User and URL, meaning one user can have multiple shortened URLs.

5. Approach to the Solution

5.1 Modular Development Strategy

The application was developed in stages:

1. Database model design
2. Utility functions
3. Core Flask setup
4. Authentication module
5. URL shortening logic
6. Analytics and enhancements
7. Frontend styling
8. Testing and validation

This step-by-step approach ensured stability and easier debugging.

5.2 URL Validation

To prevent invalid input, Python's urlparse module was used to verify:

- Presence of scheme (http or https)
- Presence of domain name (netloc)

This validation ensures only legitimate URLs are processed and stored.

5.3 Short URL Generation

Short URLs are generated using random alphanumeric strings. The application also supports **custom short URLs**, allowing users to define their own short codes, provided they are unique. Backend checks ensure no duplication occurs.

5.4 Authentication and Security

Authentication is implemented using:

- Username and password validation
- Password hashing using **bcrypt**
- Secure session management

Passwords are never stored in plain text, ensuring basic security best practices. Sessions are configured with an expiry time to automatically log users out after inactivity.

5.5 Session Management

Flask's session mechanism is used to maintain user login state. Sessions are marked as permanent and configured with a timeout duration. This prevents unauthorized access and improves application security.

5.6 Click Analytics

Each time a shortened URL is accessed, the application increments a click counter in the database. This provides basic analytics and demonstrates real-time database updates.

6. ORM Implementation

SQLAlchemy ORM was used to:

- Define database models
- Insert new records
- Query user-specific URLs
- Update click counts

An additional Jupyter Notebook (orm_sqlalchemy.ipynb) was created to demonstrate ORM operations independently of the Flask app. This notebook showcases table creation, data insertion, and querying operations.

7. Testing Strategy

The application was tested using multiple real-world scenarios:

- Valid and invalid user signup
- Login authentication failures
- Session expiry handling
- URL validation
- Duplicate custom short URLs
- URL redirection
- Click count increments
- User-specific history isolation

Testing ensured correctness, stability, and security.

8. Challenges Faced

8.1 Database Synchronization

Ensuring that the Flask app and ORM notebook used the same database required careful handling of application context.

8.2 Session Expiry

Managing session expiration correctly without affecting usability required proper Flask configuration.

8.3 Custom Short URLs

Preventing collisions while allowing custom short codes required both database constraints and backend validation logic.

9. Future Enhancements

Possible improvements include:

- Password reset functionality
- Email verification
- Role-based access control
- URL expiration
- Advanced analytics dashboard
- Deployment to cloud platforms

10. Conclusion

The Advanced URL Shortener Web Application successfully demonstrates backend development concepts such as authentication, ORM-based database management, session handling, input validation, and analytics. By extending a basic URL shortener with advanced features, the project closely resembles real-world web applications.

This project significantly enhanced understanding of Flask architecture, SQLAlchemy ORM, and secure backend design. It provides a strong foundation for building scalable and secure web applications in the future.