URL Shortener Web Application

Technical Report and Implementation Guide

Date: January 30, 2026

Institution: Data Science with Advanced GENAI Internship Nov 2025

Project Type: Full-Stack Web Development

Technology Stack: Flask (Python), SQLAlchemy ORM, SQLite, HTML5, CSS3, Bootstrap 5, JavaScript

---

Executive Summary

This report documents the development of a **URL Shortener Web Application**, a modern tool designed to convert long, complex URLs into short, shareable links. The application consists of a responsive frontend built with HTML5, CSS3, and Bootstrap 5, and a robust backend powered by Flask with SQLAlchemy ORM for database management.

The primary objective of this project is to create a user-friendly platform that allows users to:

- Shorten lengthy URLs into 6-character codes
- Store URL mappings in a persistent database
- View complete history of shortened URLs
- Copy shortened URLs with a single click
- Track click statistics for each shortened URL

The project successfully demonstrates full-stack web development principles including REST API design, database normalization, input validation, and responsive UI/UX design.

Project Objectives and Requirements

Functional Requirements

URL Shortening Engine

- Accept long URLs from users
- Generate unique 6-character short codes
- Handle duplicate URL requests efficiently
- Support both HTTP and HTTPS protocols

User Interface

- Home page for URL input and shortening
- History page displaying all previous URLs
- Copy functionality for quick URL sharing

- Clear history option for data management
- Responsive design for mobile and desktop

Data Persistence

- Store original and shortened URLs
- Maintain creation timestamps
- Track click counts for analytics
- Support bulk deletion operations

URL Validation

- Validate URL format before processing
- Automatically add HTTPS scheme if missing
- Provide meaningful error messages
- Prevent invalid URLs from being stored

Non-Functional Requirements

1. **Performance:** Fast URL shortening with minimal latency
2. **Scalability:** Support for thousands of URL mappings
3. **Security:** Input validation and XSS prevention
4. **Usability:** Intuitive interface with clear visual feedback
5. **Reliability:** Graceful error handling and recovery

System Architecture Overview

The application follows a three-tier architectural pattern:

Component	Technology	Purpose
Frontend	HTML5, CSS3, Bootstrap 5	User interface and interaction
Backend	Flask Framework	API endpoints and business logic
ORM	SQLAlchemy	Database abstraction and queries
Database	SQLite	Persistent data storage
Validation	urllib.parse, regex	URL format validation
Runtime	Python 3.8+	Application execution

Table 1: Key Components and Technologies

Design Approach

Architectural Decisions

Flask Framework Selection

Flask was chosen for this project because of its lightweight and modular design, excellent REST API development capabilities, simple routing and request handling, and suitability for educational projects. The framework's simplicity makes it ideal for rapid prototyping while maintaining code quality.

SQLAlchemy ORM Selection

SQLAlchemy provides database-agnostic design, type-safe query construction, automatic relationship handling, and straightforward migration possibilities from SQLite to PostgreSQL/MySQL. This makes the application future-proof and maintainable.

SQLite Database Selection

SQLite requires no external server, uses file-based portable storage, requires zero configuration, and is suitable for small to medium projects. This makes it ideal for development and testing environments.

Design Patterns Applied

The application follows the modified MVC (Model-View-Controller) pattern with proper separation of concerns:

- **Model:** SQLAlchemy ORM models (URLMapping class)
- **View:** HTML templates with Bootstrap styling
- **Controller:** Flask routes handling business logic

RESTful API endpoints are implemented as follows:

1. `POST /api/shorten` - Creates shortened URLs
2. `GET /api/history` - Retrieves all URLs
3. `DELETE /api/delete/<id>` - Removes specific URLs
4. `DELETE /api/clear-history` - Performs bulk deletion
5. `GET /s/<short_code>` - Redirects to original URL

Implementation Details

Project Structure

The project is organized as follows:

```
url-shortener/
├── app.py # Main Flask application
├── requirements.txt # Python dependencies
├── templates/
│ ├── index.html # Home page
│ ├── history.html # History page
│ ├── 404.html # Not found page
│   └── error.html # Error page
├── url_shortener.db # SQLite database
└── submission.zip # Final submission
```

Setup Instructions

Installation Steps

Create virtual environment

python -m venv venv

Activate virtual environment

source venv/bin/activate # macOS/Linux
venv\Scripts\activate # Windows

Install dependencies

pip install flask flask-sqlalchemy requests

Run application

python app.py

The application runs at `http://localhost:5000` after startup.

Database Design

Entity-Relationship Structure

The database uses a single table to store URL mappings with the following schema:

Column	Type	Purpose
id	INTEGER PK	Unique identifier
original_url	VARCHAR(2048)	Full original URL
shortened_url	VARCHAR(10) UQ	6-character short code
created_at	TIMESTAMP	Creation timestamp
click_count	INTEGER	Number of redirects

Table 2: URLMapping Table Schema

The table includes indexes on `shortened_url` and `created_at` for optimized query performance. The `shortened_url` column has a unique constraint to prevent duplicate short codes.

Database Normalization

The design adheres to Third Normal Form (3NF):

1. All columns contain atomic values (First Normal Form)
2. All non-key attributes depend on the entire primary key (Second Normal Form)
3. No transitive dependencies exist (Third Normal Form)

Frontend Implementation

Home Page Features

The home page (`index.html`) provides:

- URL input field with real-time validation
- Shorten button to initiate API calls
- Result display showing shortened URLs
- Copy button for clipboard functionality
- Loading states for user feedback
- Error handling with user-friendly messages

The interface uses Bootstrap 5 for responsive design, Font Awesome for icons, and vanilla JavaScript for AJAX interactions.

History Page Features

The history page (`history.html`) displays all shortened URLs with:

- Table showing original and shortened URLs
- Clickable shortened links for testing
- Timestamps indicating creation date
- Click count showing redirect statistics
- Copy and Delete action buttons
- Clear All option for bulk deletion
- Empty state message when no URLs exist

Responsive Design

The application implements mobile-first CSS with media queries for tablets and desktop devices. Touch-friendly button sizes (44px minimum) ensure usability across all devices. Flexbox-based layouts adapt automatically to different screen sizes.

Backend Implementation

Core Functions

URL Shortening Logic

The application generates random 6-character codes using alphanumeric characters. With 62 possible characters (letters and digits), this provides $62^6 = 56.8$ billion possible combinations, making collisions extremely unlikely for reasonable usage volumes.

URL Validation Strategy

URLs are validated through multiple layers:

1. Client-side validation provides immediate feedback
2. Server-side validation ensures comprehensive format checking
3. Database constraints prevent invalid data storage

The validation logic automatically adds HTTPS protocol if missing and verifies that the URL contains valid scheme and domain components.

Deduplication Approach

When users request to shorten a URL, the system first checks if that exact URL has been shortened before. If found, the existing short code is returned. Otherwise, a new unique code is generated and stored, preventing duplicate database entries while ensuring consistent mappings.

API Endpoints

POST /api/shorten

This endpoint accepts a URL in JSON format and returns a shortened URL along with the generated short code.

Request example:

```
{
"url": "https://www.example.com/very/long/url"
}
```

Response example:

```
{
"success": true,
"shortened_url": "http://localhost:5000/s/AbC123",
"short_code": "AbC123",
"message": "New shortened URL created"
}
```

GET /api/history

Returns all stored URL mappings with their metadata:

```
{
"success": true,
"data": [
{
"id": 1,
"original_url": "https://example.com",
"shortened_url": "AbC123",
"created_at": "2026-01-30 13:45:20",
"click_count": 5
}
```

]
}

DELETE /api/delete/<id> and **DELETE /api/clear-history**

These endpoints remove individual URLs or clear all stored URLs respectively, with appropriate JSON responses indicating success or failure.

GET /s/<short_code>

This endpoint redirects users to the original URL and automatically increments the click counter for analytics tracking.

Error Handling

The application implements custom error handlers for 404 (not found) and 500 (internal server) errors. Input validation prevents invalid URLs from being processed, and meaningful error messages guide users toward correct usage.

URL Validation Strategy

Multi-Layered Validation Approach

The application implements validation at three distinct layers:

Layer 1: Client-Side Validation

HTML5 `required` attribute provides immediate user feedback before server submission.

Layer 2: Server-Side Validation

Python validation using `urllib.parse` examines URL structure, scheme, and domain components. Missing HTTPS protocol is automatically added for convenience.

Layer 3: Database Constraints

Unique constraints on the `shortened_url` column and not-null constraints on required fields ensure data integrity at the database level.

URL Component Parsing

The validation function parses URLs into components: scheme (protocol), netloc (domain), path, query parameters, and fragment identifiers. Each component is verified to ensure the URL is well-formed before storage.

Testing and Deployment

Testing Strategy

Unit Testing

Testing functions independently ensures correct behavior:

1. Short code generation produces 6-character alphanumeric strings
2. URL validation correctly identifies valid and invalid formats
3. Deduplication returns same code for identical URLs

Integration Testing

Complete workflow testing verifies:

1. URL shortening endpoint creates new entries
2. History endpoint returns all stored URLs
3. Redirect endpoint correctly increments click count
4. Delete operations remove URLs as expected

Deployment Considerations

For local development, the application runs with `python app.py` at `http://localhost:5000`.

For production deployment, Gunicorn is recommended:

```
pip install gunicorn
gunicorn -w 4 -b 0.0.0.0:8000 app:app
```

Database migration from SQLite to PostgreSQL is straightforward due to SQLAlchemy's database-agnostic design. Only the connection string requires updating.

Future Enhancements

Planned Features

Phase 2: User Accounts

- User registration and authentication
- Personal URL management
- Privacy settings (public/private URLs)
- Per-user analytics

Phase 3: Analytics

- Detailed click statistics
- Geographic data tracking
- Device and browser information
- Referrer tracking

Phase 4: Advanced Features

- Custom short codes
- QR code generation
- Bulk URL shortening

- API key management
- Webhook notifications

Phase 5: Performance Optimization

- Redis caching for frequently accessed URLs
- CDN integration for faster redirects
- Database replication for high availability
- Load balancing across multiple servers

Security Enhancements

Rate limiting prevents abuse of the API endpoints. CSRF protection tokens secure form submissions. The application already implements SQL injection prevention through SQLAlchemy's parameterized queries and XSS prevention through Jinja2's template escaping.

Conclusion

Project Summary

This URL Shortener Web Application successfully demonstrates a complete full-stack web development project encompassing responsive frontend design, RESTful backend API, efficient database design, and comprehensive URL validation.

Key Achievements

1. ✓ Shortened URL generation using 6-character unique codes
2. ✓ Persistent storage with SQLite and SQLAlchemy ORM
3. ✓ URL history tracking with timestamps and click counts
4. ✓ Comprehensive multi-layer URL validation
5. ✓ Responsive design for mobile and desktop devices
6. ✓ RESTful API architecture with proper error handling
7. ✓ Copy-to-clipboard functionality
8. ✓ Bulk deletion capabilities
9. ✓ Redirect tracking with analytics
10. ✓ Clean and maintainable code structure

Technical Highlights

The application achieves excellent performance through indexed database queries, handles deduplication efficiently with 56.8 billion possible combinations, and provides responsive user experience with instant feedback. The scalable architecture can handle thousands of URLs efficiently while maintaining security through input validation and XSS prevention.

Learning Outcomes

This project demonstrates competency in full-stack web development, web frameworks (Flask), database design and ORM usage (SQLAlchemy), RESTful API development, responsive frontend technologies (HTML5, CSS3, Bootstrap, JavaScript), form handling and validation, and problem-solving through algorithm design and data structure optimization.

The modular architecture enables independent scaling of components, and the clean code structure facilitates future maintenance and feature additions. The project serves as an excellent foundation for learning web development concepts and is production-ready for practical deployment.

Report Generated: January 30, 2026

Project Status: Complete and Fully Functional ✓