URL Shortener Web Application (Basic Version)

1. Introduction

As a participant in the Data Science with Advanced GENAI Internship at Innomatics Research Labs, I was assigned the task of developing a web application from scratch. The objective of this specific assignment was to build a "Basic URL Shortener Web Application." Long URLs are often cumbersome, difficult to read, and hard to share efficiently. The core goal of this project was to create a functional system that takes a lengthy web address, converts it into a concise, easily shareable link, and properly redirects users to the original destination when the short link is clicked.

When I reviewed the project brief, I paid close attention to the strict rules and requirements provided. I needed to ensure my implementation perfectly matched the mandated technology stack while delivering a seamless user experience. This report documents my complete, step-by-step approach to building the application, the challenges I encountered, and how I successfully implemented the required features to meet the assignment guidelines.

2. Required Technology Stack

To ensure my project adhered strictly to the guidelines set by Innomatics Research Labs, I utilized the exact technology stack mandated in the assignment brief. I did not deviate from these requirements:

- **Frontend (HTML, CSS, and Bootstrap):** As instructed, I built the user interface using HTML and CSS, integrating the Bootstrap framework. Using Bootstrap allowed me to efficiently implement responsive navigation bars, styled input forms, buttons, and structured data tables without relying on external UI libraries outside of the project scope.
- **Backend (Flask - Python):** Following the backend requirements, I utilized Python along with the Flask web framework. Flask provided the necessary routing capabilities to handle HTTP GET and POST requests, linking the frontend forms to the backend logic perfectly.
- **Database & ORM (SQLite and SQLAlchemy):** The guidelines strictly required the use of a Database ORM and an SQLite database. To fulfill this, I integrated Flask-SQLAlchemy. This allowed me to map my Python classes directly to an SQLite

database file, handling data storage securely and efficiently without writing raw, complex SQL queries.

3. Step-by-Step Implementation

Phase 1: Designing the Database Architecture

My first step was establishing the data structure using the required ORM. I needed a simple, efficient way to map a user's long URL to the short code my application would generate.

Using SQLAlchemy, I created a database model named URLMap. I designed this table with three specific columns:

1. **id**: An integer-based primary key that auto-increments for every new entry.
2. **original_url**: A string column designed to store the destination web address pasted by the user.
3. **short_id**: A string column that stores the randomly generated 6-character short code. Crucially, I set unique=True for this column to guarantee that the database would never accidentally assign the exact same short link to two different websites.

Phase 2: Building the Core Backend Logic

With the database structure ready, I moved on to the backend logic. I had to write Python functions to handle the core mechanics of the URL shortening process.

- **Generating the Short Link:** I imported Python's built-in string and random modules. I wrote a helper function that randomly selects six characters from a pool of uppercase letters, lowercase letters, and digits. This provides millions of possible unique combinations for the short links.
- **Routing & Redirection:** I created a dynamic route in my Flask application (/<short_id>). The logic here is straightforward: when a user navigates to a short link generated by the app, the backend extracts that specific 6-character ID, queries the SQLite database to find the matching original_url, and uses Flask's redirect() function to route the user to their intended destination.

Phase 3: Fulfilling Specific Assignment Rules

The project guidelines included specific functional requirements that required careful implementation:

- **Rule: URL Verification:** The instructions explicitly stated: *"Try to verify whether the URL entered by the user is correct or not. (Do some googling to find out how to make it possible)"*. To achieve this, I researched and utilized the Python requests library. I implemented a function that sends a quick HEAD request to the URL the user submits. If the website does not exist, or if it returns an error status code, my application catches the exception and flashes an error message on the screen, successfully preventing the creation of broken short links.
- **Rule: The Copy Button:** The user needed to be able to copy the generated link easily. I wrote a client-side JavaScript function triggered by a "Copy" button. This function selects the text inside the short URL input box, utilizes navigator.clipboard.writeText() to copy it to the clipboard, and triggers a browser alert confirming the action, exactly as required.

Phase 4: Developing the Frontend Interface

I structured the frontend into two distinct pages, connected by a unified Bootstrap navigation bar, directly satisfying the project's page requirements:

1. **The Home Page:** I designed a clean, centralized card interface. Users input their long URL into a form. Upon submission, the backend generates the short link, and the page dynamically re-renders to display the shortened URL in a read-only text field right next to the functional "Copy" button.
2. **The History Page:** I created a dedicated route that queries all stored records in the URLMap database table. I passed this data to the HTML template using Jinja2 templating. The data is displayed in a styled Bootstrap table, presenting every original URL side-by-side with its clickable short link, ensuring complete transparency for the user.

4. Challenges Faced and Solutions

During the development process, I encountered a few technical challenges that required refinement:

1. **Handling Missing HTTP Protocols:** I realized that users frequently type domain names (e.g., github.com) without the http:// or https:// prefix. This broke the redirection process, as the browser interpreted it as a local route. To solve this, I added a backend preprocessing step: if the submitted string lacks these protocols, the application automatically prepends http:// before validating and saving it.

2. **Preventing Duplicate Database Entries:** If a user shortened the exact same link multiple times, the database created duplicate rows. To optimize storage, I implemented a check: the app first queries the database to see if the original_url already exists. If it does, it simply returns the existing short code rather than generating a redundant one.

5. Conclusion

Developing this URL Shortener Web Application was a highly practical exercise in full-stack development. By strictly following the guidelines provided by Innomatics Research Labs, I successfully integrated a Bootstrap-powered frontend with a Flask backend, utilizing SQLAlchemy for persistent data storage.

I successfully implemented all the mandated features of the Basic Version, including real-time URL validation and a functional clipboard copy feature. This project greatly enhanced my understanding of HTTP request handling, database ORM integration, and user-centric web design.