NAME: NIKHIL BAGUL INTERNID: 277

roof of Concept: Web-Based URL Shortener

1. Introduction & Objective

1.1. Problem Statement

In the digital landscape, sharing links is a fundamental activity. However, many URLs, especially those with tracking parameters or deep-linking information, can be excessively long, cumbersome, and aesthetically unpleasing. These long URLs are difficult to share in character-limited environments (like Twitter/X), hard to remember, and can break when copied and pasted.

1.2. Project Goal

The primary goal of this project is to design, develop, and document a functional, web-based URL shortener application. This tool will serve as a proof of concept (POC) to demonstrate the core functionality of taking a long URL as input and providing a short, unique, and persistent alias. When this alias is accessed, the user will be seamlessly redirected to the original destination URL. This service is analogous to popular commercial services like bit.ly and TinyURL.

2. Scope of Work

2.1. In-Scope Features

The scope of this POC is strictly limited to the core functionality required to prove the concept's viability.

- **User Interface:** A minimalist, single-page web interface containing an input field for the long URL and a submission button.
- **URL Shortening:** A backend process to generate a unique, non-sequential alphanumeric "slug" of 8 characters for each submitted URL.
- **URL Mapping & Storage:** A persistent storage mechanism (database) to store the direct mapping between the original long URL and its generated short slug.
- **Redirection Service:** A backend mechanism that listens for requests to shortened URLs, looks up the original URL in the database, and performs an HTTP 301 (permanent) redirect.
- **API Endpoint:** A simple REST API with one primary endpoint (/shorten) to handle the creation of new short URLs.
- **Basic Error Handling:** The system will handle fundamental errors, such as invalid URL submissions and requests for non-existent short links.

2.2. Out-of-Scope Features

To maintain focus on the core objective, the following features will be explicitly excluded from this POC:

- User accounts, authentication, and link history.
- Customizable or branded short links.
- Advanced analytics (e.g., click tracking, geographic data, referral sources).

- Link expiration, password protection, or one-time-use links.
- A dedicated administrative dashboard for managing links.

3. System Architecture & Technology Stack

The application is built on a modern, decoupled client-server model.

3.1. Frontend (Client)

- **Technology:** HTML5, Tailwind CSS, and vanilla JavaScript.
- **Rationale:** A static HTML/CSS/JS frontend is lightweight, fast, and simple to deploy. Tailwind CSS is used for rapid, utility-first styling. Vanilla JavaScript is sufficient for the simple API interactions required, avoiding the overhead of a larger framework like React or Vue for this POC.

3.2. Backend (Server)

- **Technology:** Node.js with the Express.js framework.
- **Rationale:** Node.js offers an event-driven, non-blocking I/O model, making it highly efficient for handling concurrent requests, which is ideal for an API-driven application. Express.js provides a minimal and flexible set of features for building a robust web server and REST API.

3.3. Database

- **Technology:** Google Firestore (a NoSQL, document-based database).
- **Rationale:** Firestore is a fully managed, scalable database that integrates seamlessly with a Node.js backend. Its document-based model is a natural fit for storing URL mappings. Using the short slug as the document ID allows for extremely fast and efficient lookups (O(1) complexity), which is critical for the redirection service.

4. Implementation Details & User Flow

4.1. Slug Generation Strategy

A core component of the system is the generation of the short, unique slug. We will use the nanoid library. This approach is chosen over alternatives for several key reasons:

- **Uniqueness:** nanoid has an extremely low probability of generating duplicate strings.
- **Non-Sequential:** Unlike using an auto-incrementing database ID, the generated slugs are not predictable, which is a minor security benefit as it prevents users from guessing other short links by incrementing a number.
- **URL-Friendly:** It generates URL-friendly characters by default.

4.2. Database Schema

The Firestore database will contain a single collection named urls. Each document within this collection represents a single shortened link. The structure is as follows:

- **Collection:** urls
    - **Document ID:** The unique 8-character slug (e.g., aB1cD2eF).

- **Fields:**
    - longUrl (string): The original URL the user submitted.
    - slug (string): A copy of the slug, for potential future querying needs.
    - createdAt (timestamp): The date and time the link was created.

4.3. Detailed User Flow: Shortening a URL

1. **User Interaction:** The user navigates to the web application, pastes a long URL into the input form, and clicks the "Shorten URL" button.
2. **Frontend Action:** The client-side JavaScript prevents the default form submission. It retrieves the value from the input field and sends an asynchronous POST request to the backend's /shorten endpoint. The request body contains a JSON object: { "longUrl": "..." }.
3. **Backend Processing:** a. The Express server receives the POST request. b. It validates the longUrl from the request body to ensure it is a properly formatted URL. c. It calls nanoid(8) to generate a unique slug. d. It connects to Firestore and creates a new document in the urls collection, using the generated slug as the document ID and saving the longUrl and other metadata.
4. **Backend Response:** The server responds to the frontend with a 201 Created status code and a JSON object containing the full short URL: { "shortUrl": "http://localhost:3000/aB1cD2eF" }.
5. **Frontend Display:** The client-side JavaScript receives the successful response, parses the JSON, and dynamically updates the DOM to display the returned shortUrl to the user, along with a "Copy" button.

4.4. Detailed User Flow: Accessing a Shortened URL

1. **User Interaction:** The user clicks or enters a short URL (e.g., http://localhost:3000/aB1cD2eF) into their browser.
2. **Backend Processing:** a. The Express server receives a GET request for the path /:slug. b. It extracts the slug (aB1cD2eF) from the request parameters. c. It queries the urls collection in Firestore for a document with an ID matching the extracted slug.
3. **Redirection Logic:**
    - **If Found:** The backend retrieves the longUrl field from the document. It then issues an HTTP 301 Moved Permanently redirect response to the browser, with the Location header set to the value of the longUrl. The user's browser handles the rest.
    - **If Not Found:** The backend responds with a 404 Not Found status code and a JSON error message.

5. Error Handling

The system will manage the following error conditions:

- **Invalid URL Submission:** If the longUrl provided in a POST request is not a valid URL format, the server will respond with a 400 Bad Request status and an error message.

- **Slug Not Found:** If a GET request is made to a slug that does not exist in the database, the server will respond with a 404 Not Found status.
- **Server Errors:** Any unexpected database or server errors will be caught, and a generic 500 Internal Server Error will be returned to the client.