URL Shortener Application Design Document

---

1. Overview

This document outlines the architecture and design decisions for building a scalable and reliable URL shortener application. The application consists of a frontend built using Angular 16 and a backend API built using .NET Core 8. It is designed to allow users to shorten URLs, retrieve the original URLs, and track analytics like click counts.

---

2. Architecture

The architecture of the application follows a microservices-based approach. The frontend communicates with the backend API, which handles all the business logic, including URL shortening, redirection, and analytics.

Components:

1. **Frontend:**
    - **Framework:** Angular 16
    - **Responsibilities:**
        - Create a user interface for shortening URLs.
        - View URL analytics (click counts and redirection details).
        - Handle user interactions and request redirection or shortening operations from the backend.
2. **Backend:**
    - **Framework:** .NET Core 8
    - **Responsibilities:**
        - Handle all incoming requests (URL shortening, redirection, and analytics).
        - Perform URL shortening and store the mapping in the database.
        - Track click counts and provide analytics via the /analytics endpoint.
3. **Database:**
    - **SQL Server** for storing URL mappings and click analytics.
    - **Entities:**

1. **Urls**: Stores the original URL and its corresponding shortened code.
2. **Analytics**: Tracks the number of times a shortened URL is accessed.

---

3. Technology Stack

- **Frontend:**
    - Angular 16 for creating the web interface.
    - Bootstrap and CSS for responsive design and a clean UI.
- **Backend:**
    - **.NET Core 8** for building the RESTful API.
    - **Dapper** for efficient ORM and database interactions.
    - **SQL Server** for data persistence and scalable storage.
- **Additional Tools:**
    - **Postman** for API testing and debugging.
    - **Git** for version control and code management.
    - **Docker (optional)** for containerization and simplifying deployment processes.

---

4. Database Design

The database will store two main entities: **URLs** and **Analytics**.

1. URLs Table:

This table stores the mapping between the original URL and the shortened URL.

- **Columns:**
    - **ShortenedCode (Primary Key):** The unique shortened code.
    - **LongUrl:** The original long URL.
    - **CreatedAt:** Timestamp of when the URL was shortened.
    - **ShortenedCode :**The shortened code for the URL.
    - **ClickCount:** The number of times the shortened URL has been accessed.

---

5. API Design

The API exposes the following endpoints for interaction with the frontend.

1. POST /api/shorten:

- **Description:** Accepts a long URL and returns a shortened URL.
- **Request Body:**

json

Copy code

{ "longUrl": "http://example.com" }

- **Response:**

json

Copy code

{ "shortenedUrl": "http://localhost/abc123" }

2. GET /api/redirect/{shortenedUrl}:

- **Description:** Redirects the user to the original URL.
- **Request Parameter:** shortenedUrl (the code part of the shortened URL).
- **Response:** Redirects the user to the original long URL.

3. GET /api/analytics:

- **Description:** Fetches analytics data (click count) for a shortened URL.
- **Request Parameter:** shortenedUrl (the full or part of the shortened URL).
- **Response:**

json

Copy code

{ "shortenedUrl": "http://localhost/abc123", "clickCount": 100 }

---

6. Security

The application ensures secure URL shortening and redirection by implementing the following measures:

1. **Input Validation:**
    - Ensures that only valid URLs are accepted (e.g., correct format, no potential for malicious inputs).
    - Prevents potential SQL injection attacks by using parameterized queries.
2. **HTTPS:**
    - All communication between the frontend, backend, and database is encrypted using HTTPS, ensuring secure transmission of data.
3. **API Authentication:**
    - **OAuth 2.0 or API Keys** (Optional): To secure the API endpoints, ensuring that only authorized clients can make requests (especially for analytics and URL shortening operations).
    - **Rate Limiting:** To prevent abuse, limit the number of requests a client can make to the API in a specified time period (e.g., 1000 requests per hour).
4. **Data Encryption:**

- Sensitive information, such as user data and analytics, should be encrypted at rest and in transit.
- Passwords (if implemented) should be hashed using strong cryptographic algorithms such as SHA-256.

---

7. Scalability and Performance Considerations

1. **Load Balancing:**
   - Distribute incoming traffic across multiple instances of the backend using a load balancer to ensure high availability and better performance.
2. **Database Optimization:**
   - Use **indexing** on frequently queried columns (e.g., ShortenedCode in the **URLs** table) to speed up lookups.
   - **Sharding** can be applied for horizontally scaling the database as the number of URLs and requests grows.
3. **Caching:**
   - Implement caching mechanisms (e.g., Redis) for frequently accessed URLs to reduce the load on the database and improve response times.
4. **Asynchronous Processing:**
   - For tasks such as click tracking and analytics updates, use background processing (e.g., using a message queue like RabbitMQ) to handle non-blocking operations.

---

8. Future Enhancements

1. **Custom Short URLs:**
   - Allow users to specify custom aliases for shortened URLs, giving them control over the shortened code.
2. **User Authentication:**
   - Introduce user accounts and profiles so users can track their shortened URLs, view analytics, and manage their links.
3. **Advanced Analytics:**
   - Provide more detailed analytics such as geographic location, device types, referrer websites, and more to help users understand how their links are being used.
4. **Social Media Integration:**
   - Enable users to share shortened URLs directly on social media platforms, driving engagement with the service.

---

9. Conclusion

This design provides a solid foundation for building a highly scalable, secure, and reliable URL shortener application. By following best practices for security, performance, and user experience, the application will be capable of handling a high volume of requests while ensuring the safety and privacy of its users.