Architecture Documentation: Blog Application

1. Application Overview

The Blog Application allows users to create blog posts and comment on existing posts. The application has been divided into two microservices:

- **Post Service**: Handles blog post creation and management.
- **Comment Service**: Manages comments on blog posts.

The application was originally built as a monolithic application but has been refactored into microservices for better scalability, flexibility, and deployment.

2. Before: Monolithic Architecture

In the initial version of the application, all features and functionalities were packaged into a single Flask app. This included:

- **Blog Post Management**: Create, view, update, and delete posts.
- **Comment Management**: Add, view, and manage comments.

Diagram (Monolithic Architecture)

Blog Application (Flask App)
- Post Management
- Comment Management

All the functionalities existed within a single application, meaning that every component depended on the same Flask app and database.

3. After: Microservices Architecture

In the refactored version, the monolithic app has been divided into two separate microservices, each responsible for a specific set of functionalities.

- **Post Service**:
  - Handles creating, updating, retrieving, and deleting blog posts.
  - Stores data in its own **SQLite** database.
- **Comment Service**:
  - Manages to add and view comments for specific blog posts.
  - Stores comment data in its own **SQLite** database.

Each microservice runs in its own Docker container, and communication between the services is done via HTTP REST API.

Diagram (Microservices Architecture)

Post Service	Comment Service
- Flask-based service - Manages posts (CRUD) - SQLite (post.db)	- Flask-based service - Manages comments (CRUD) - SQLite (comment.db)
HTTP REST API Communication	

4. Communication Between Microservices

The two services communicate via **HTTP REST APIs**.

- **Post Service** exposes endpoints such as:
  - `GET /posts`: Retrieve all posts.
  - `POST /posts`: Create a new post.
  - `DELETE /posts/<id>`: Delete a post by ID.
- **Comment Service** exposes endpoints such as:
  - `GET /comments/<post_id>`: Retrieve comments for a specific post.
  - `POST /comments`: Add a comment to a post.
  - `DELETE /comments/<comment_id>`: Delete a comment by ID.

There is no direct database sharing between the services. Each microservice manages its database independently, promoting loose coupling between them.

5. Kubernetes Deployment

The application is deployed on a **Kubernetes cluster** using **kind** (Kubernetes in Docker). Each microservice is deployed as a separate pod, with corresponding services exposed via **NodePort** for external access.

Kubernetes Architecture

Post Pod	Comment Pod
- Flask-based Post Service - SQLite (post.db) - Kubernetes Deployment	- Flask-based Comment Service - SQLite (comment.db) - Kubernetes Deployment
Post Service (NodePort)	| Comment Service (NodePort)

6. Benefits of Microservices Architecture

- **Scalability**: Each service can be independently scaled based on traffic.
- **Independent Deployment**: Changes to one service can be deployed without affecting the other.
- **Fault Isolation**: If one service fails, it doesn't bring down the entire application.
- **Technology Flexibility**: Different technologies or databases could be used for each microservice.

7. Conclusion

The transition from a monolithic to microservices-based architecture has improved the scalability, flexibility, and maintainability of the Blog Application. By leveraging Docker and Kubernetes, the application can be easily deployed and managed in a cloud-native environment.