# System Design and Architecture

## 1. Modular Design & System Architecture

From the very beginning, we built the app using a modular approach. Our system is divided into two main, independent parts: the Front-End and the Back-End.

**System Components:**

- **Front-End (The Flutter App):** This is what runs on the user's phone. Even this part is modular:
    - **UI Layer (The "Views"):** These are all the individual screens you've created (signup.dart, login.dart, author_dashboard.dart). Their only job is to look good and show information to the user.
    - **Service Layer :** This is our services folder, which contains auth.dart, database.dart, and storage.dart. This layer is the real workhorse. It handles all the logic, like how to log a user in, how to save a post, or how to upload an image. The UI Layer talks to this Service Layer to get things done.
- **Back-End (Firebase):** This is our powerful, cloud-based engine.
    - **Firebase Authentication:** Acts as our app's security guard, managing all user signups and logins.
    - **Firestore Database:** This is our organized filing cabinet for all the text-based data, like user profiles, roles, and post details.
    - **Firebase Storage:** This is our bulk storage warehouse, specifically for large files like user-uploaded images.

## System Architecture Diagram:

Here is a simple flowchart that shows how all these modules work together when an author posts an announcement:

1. **Author** interacts with the **UI Layer**.
2. The UI Layer calls a function in the **Service Layer**.
3. The Service Layer securely communicates with the **Firebase Back-End**.
4. Data flows back to the app, confirming the post was successful.

**Why This Modular Approach is Better:**

- **Easier to Maintain:** If we ever want to change how we handle image uploads, we only need to edit one file (storage.dart), not every single page that has an upload button.
- **Reusable Code:** Our AuthService is a perfect example. We wrote it once, and now both the signup.dart and login.dart pages can use it. We didn't have to write the same login code twice.
- **Ready for Growth:** This design makes it very easy to add new features. If we want to add notifications, we can just create a new notification_service.dart file without breaking anything else.

## 2. Technology Stack

- **Front-End Framework:** Flutter (with Dart)
- **Back-End Platform:** Google Firebase
    - **Firestore Database:** I  chose Firestore because it's a real-time database. This is the magic that allows the "My Content" page to update instantly after an author publishes a new post, creating a very dynamic user experience.
    - **Firebase Authentication:** This was a huge time-saver. It provides a secure, ready-made solution for handling user accounts, including email/password and Google Sign-In. Building this from scratch would have taken weeks.
    - **Firebase Storage:** When it comes to storing user-uploaded files like images, Firebase Storage is the perfect tool. It's built to handle large files and integrates directly with our Firestore database and security rules.

## 3. Scalability Planning

- **How We Handle More Users:** Our choice of **Firebase** is the core of our scalability plan. Firebase is a Google product and is designed to scale automatically. As more users sign up and create content, Firebase seamlessly allocates more resources behind the scenes. We don't have to worry about buying new servers or managing database capacity; it's all handled for us.

- **Addressing Potential Bottlenecks:**
    1. **Slow Database Queries:**
        - **Problem:** As our app grows to have thousands of books and posts, searching for specific content could become slow.
        - **Solution:** We have already implemented the primary solution: **Firestore Indexing**. By creating indexes on our data (for example, on the createdAt field), we give Firestore a "cheat sheet" to find the information it needs almost instantly, even in a huge database.
    2. **Slow App Performance on Bad Internet:**
        - **Problem:** Users with slow or unreliable internet connections might have a bad experience waiting for content to load.
        - **Solution:** Firestore has a fantastic built-in feature for **offline persistence**. It automatically saves a copy of recently viewed data on the user's device. This means that pages like "My Content" can load instantly from the local cache, even if the user has a poor connection.
    3. **Controlling Costs:**
        - **Problem:** As our user base grows, so will the cost of using Firebase.
        - **Solution:** We are using Firebase's "Blaze" plan, which is pay-as-you-go, so our costs are directly related to our success. We can implement **budget alerts** in the Google Cloud Console to get notified if our spending is approaching a certain limit. Furthermore, by designing our database queries efficiently (using indexes and fetching only the data we need), we minimize the number of reads and writes, which directly reduces our monthly bill.