WEEK 3

1. High-Level Architecture Diagram

The high-level architecture of *CampusConnect* is a client-server model, with Flutter powering the cross-platform mobile frontend and Firebase providing scalable backend services. The diagram below outlines the key components and their interactions.

- **Mobile Client (Flutter)**: A single codebase for iOS and Android, featuring screens for Home Feed, Event Calendar, Group Chats, Marketplace, and Profile.
- **Firebase Cloud Backend**: Handles authentication, real-time data storage, serverless logic, and file storage.
- **External Services**: Includes Firebase Cloud Messaging (FCM) for push notifications and optional analytics tools.

---

2. Selected Design Patterns & Justification

The following three design patterns are selected for their suitability to *CampusConnect*'s requirements, leveraging Flutter's reactive framework and Firebase's real-time capabilities. Each is justified for its role in the system.

- **Observer Pattern**
    - **Role**: Manages real-time updates for features like group chats, event notifications, and feed posts by listening to Firestore streams and updating the UI accordingly.
    - **Justification**: *CampusConnect* relies heavily on real-time data (e.g., new messages in study groups). Flutter's StreamBuilder widget aligns perfectly with the Observer Pattern, subscribing to Firestore changes to keep the UI in sync instantly. This ensures a seamless experience for users expecting live updates, making it critical for the app's core functionality.
- **Factory Pattern**
    - **Role**: Creates objects for different types of app entities, such as posts (text, image, event) or notifications (event reminders, chat alerts).
    - **Justification**: The app handles various content types with shared behaviors but different implementations (e.g., rendering an image post vs. a text post).

A factory class (e.g., PostFactory) simplifies object creation, enhances code maintainability, and supports future extensions (e.g., adding video posts) without altering existing logic. This pattern is ideal for managing diverse data in a structured way.

- **Adapter Pattern**
    - **Role**: Converts Firestore's raw data format into Flutter-compatible data models (e.g., mapping a Firestore document to a Post object) or integrates external APIs with the app's structure.
    - **Justification**: Firebase Firestore returns data in a generic format (maps), which needs to be adapted for Flutter's typed models. The Adapter Pattern (e.g., via a FirestoreAdapter) ensures smooth data transformation, decoupling the backend from the frontend. This flexibility is essential for maintaining compatibility if Firebase's API changes or if new data sources are added.

These patterns are the most suitable because:

- **Observer** addresses the app's real-time requirements, a core feature for chats and notifications.
- **Factory** streamlines content creation, supporting the app's diverse data types.
- **Adapter** ensures seamless integration between Flutter and Firebase, enhancing maintainability.

The other patterns were less critical:

- **Builder**: While useful for complex object construction, Flutter's widget system and simple data models reduce its necessity.
- **Prototype**: Cloning is less relevant since most objects (e.g., posts, events) are created anew or fetched from Firestore.
- **Proxy**: Lazy-loading or access control can be handled by Firebase's built-in mechanisms, reducing the need for a dedicated proxy.

---

3. UML Diagrams

Component Diagram

The Component Diagram illustrates the app's software components and their interactions, incorporating the selected patterns.

- **Flutter Client**:
    - **UI Components**: Built with Flutter widgets, using Observer (via StreamBuilder) for real-time UI updates.
    - **Business Logic**: Includes Factory for creating objects (e.g., posts), Adapter for data conversion (e.g., Firestore to models), and Observer for managing data subscriptions.
- **Firebase Backend**: Provides Authentication, Firestore Database, Cloud Storage, and Cloud Functions, accessed via HTTPS.

Deployment Diagram

The Deployment Diagram shows how the system is deployed across physical nodes.

text

- **Mobile Device**: Hosts the Flutter app, running on iOS or Android, communicating with Firebase over HTTPS.
- **Firebase Cloud**: A single node hosting Authentication, Firestore, Cloud Storage, Cloud Functions, and Hosting services.

---

4. Data Flow Overview

The data flow outlines how information moves through *CampusConnect*, from user interactions to backend processing and UI updates, leveraging the Observer, Factory, and Adapter patterns.

1. **User Interaction**:
    - A student logs in via the Flutter app, triggering a request to Firebase Authentication.
    - The app's UI (e.g., Home Feed) requests data using a repository pattern enhanced by the **Adapter Pattern** to convert user inputs into API calls.
2. **Authentication**:
    - Firebase verifies the university email and returns a secure token, allowing access to protected features.
3. **Data Retrieval**:

- The app queries Firestore for data (e.g., posts, events). The **Adapter Pattern** transforms Firestore's raw maps into Flutter models (e.g., Post objects).
- The **Observer Pattern** (via StreamBuilder) subscribes to Firestore streams, ensuring real-time updates for chats or feeds.

4. **User Actions**:
   - The student creates a post (e.g., an event announcement). The **Factory Pattern** (e.g., PostFactory) generates the appropriate post type (EventPost vs. TextPost), which is sent to Firestore.
   - For group chats, messages are saved to Firestore, and the **Observer Pattern** notifies all group members, updating their UI instantly.
5. **Backend Processing**:
   - Firebase Cloud Functions handle tasks like sending push notifications (via FCM) for event reminders.
   - The **Adapter Pattern** ensures compatibility between Cloud Functions' output and Flutter's data models.
6. **Feedback to User**:
   - The Flutter app refreshes the UI with new data (e.g., a posted message) using reactive widgets tied to the **Observer Pattern**.

Example Flow for Posting in a Group Chat:

- User types a message → Flutter UI captures input → **Factory Pattern** creates a ChatMessage object → **Adapter Pattern** formats it for Firestore → Firestore saves and syncs the message → **Observer Pattern** (via StreamBuilder) updates the chat UI for all group members.