**Introduction to Asynchronous Programming in Flutter**

TEAM MEMBERS:

1. Laishram Siddarth 21csu216
2. Aditya Narain Jha 21csu129

* Explanation of synchronous vs. asynchronous programming.
* Overview of async/await, generators, futures, and streams in Dart.
* Importance of asynchronous programming in improving app responsiveness and user experience.

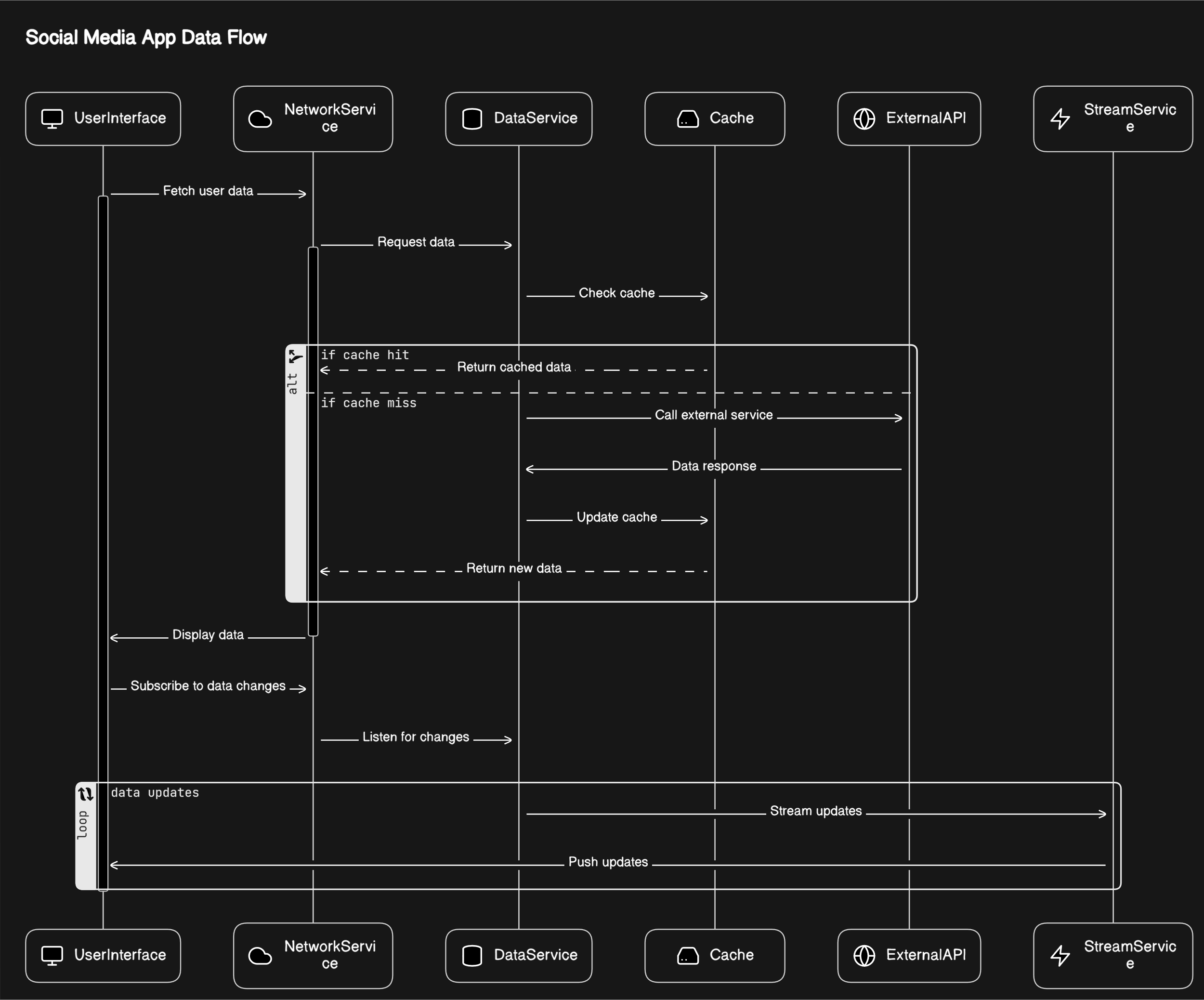
**Core Concepts and Tools**

* Detailed explanation of Futures and their role in handling asynchronous operations.
* Understanding Streams and StreamControllers for real-time data handling.
* Generators and the use of yield in creating iterable sequences.

**Implementing Async/Await in Social Media Features**

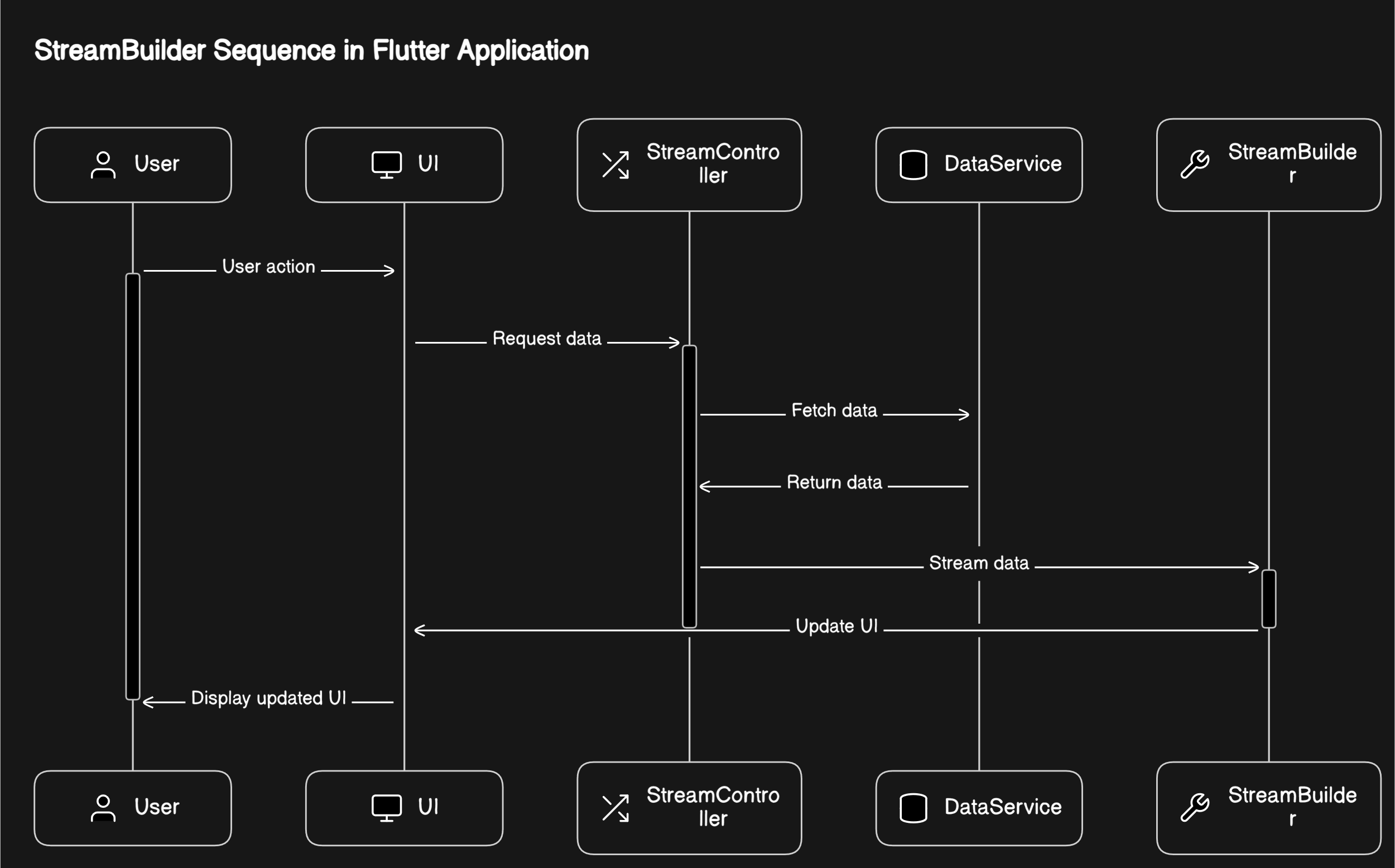
* Step-by-step guide on using async/await for network requests (e.g., fetching posts, user profiles).
* Examples of improving UI responsiveness with asynchronous operations.
* Handling errors in asynchronous operations with try-catch blocks.

Asynchronous Operation Flow in Flutter



**Utilizing Streams for Real-Time Updates**

* How to set up a StreamController to manage data flow.
* Using StreamBuilder in the Flutter UI to display real-time updates (e.g., new posts, comments).
* Best practices for managing subscriptions and avoiding memory leaks.

Stream and StreamBuilder Integration

**Error Handling and Best Practices**

* Common pitfalls in asynchronous programming and how to avoid them.
* Strategies for error handling in Futures and Streams.
* Tips for writing clean and maintainable asynchronous code in Flutter.

1. **Fetching Posts**:
   * Traditional Synchronous Method:

List<Post> fetchPosts() {

// Blocking UI while fetching posts

// Inefficient if there are a large number of posts

// May cause the app to freeze if the network is slow

// Return all posts at once }

* + Using Async/Await:

Future<List<Post>> fetchPostsAsync() async {

// Non-blocking UI while fetching posts

// Efficient, as it allows other UI operations to continue

// Uses async/await for asynchronous operations

// Returns a Future representing the result of fetching posts }

1. **Following Users**:
   * Traditional Synchronous Method:

void followUser(User user) {

// Blocking UI while following user

// May cause UI freeze if there is a delay in network operation

// Synchronously follow the user }

* + Using Futures:

Future<void> followUserAsync(User user) async

{ // Non-blocking UI while following user

// Uses async/await for asynchronous operations

// Returns a Future representing the result of following user }

1. **Real-Time Updates**:
   * Using Generators:

Stream<Post> fetchPostsStream() async\* {

// Use async generator to fetch posts one by one

// Allows for lazy loading and efficient data fetching

// Yield posts one by one as they are fetched }

* + Using Streams:

StreamController<Post> \_postStreamController = StreamController<Post>(); Stream<Post> get postStream => \_postStreamController.stream; void startFetchingPosts() {

// Start fetching posts in the background fetchPostsStream().listen((post) {

// Add each fetched post to the stream \_postStreamController.add(post); }); }

By leveraging async/await, generators, Futures, and Streams, we can make our Social Media App more responsive and efficient, ensuring a smoother user experience even during heavy network operations.

