Offline-First Architecture in Mobile Applications

Introduction: What is Offline-First?

In today's mobile world, users expect apps to be fast, reliable, and always available, regardless of network connectivity. An **Offline-First** architecture is a design philosophy where your mobile application prioritizes local data storage and processing. This means the app functions fully or largely independently of an internet connection. Network connectivity is then used primarily for *synchronization* with a remote backend, rather than being a prerequisite for core functionality.

It's not just about caching. It's about building the app as if there's *no* network, and then adding network capabilities as an enhancement for data synchronization.

1. Why Offline-First Architecture?

Implementing an offline-first strategy offers significant advantages:

- **Reliability:** The app works even with no internet, flaky connections, or slow networks. This is critical for users in areas with poor coverage (e.g., rural areas, underground, travel).
- **Performance:** Reading from a local database is significantly faster than fetching data over a network. This leads to a snappier, more responsive user experience.
- **Improved User Experience (UX):**
  - **Always Available:** Users can always access and interact with their data.
  - **No Loading Spinners:** Reduced wait times and fewer frustrating loading states.
  - **Optimistic UI:** User actions (e.g., sending a message) can be reflected instantly in the UI, even before syncing with the server.
- **Battery Efficiency:** Less reliance on constant network requests reduces battery consumption.
- **Reduced Server Load:** Fewer direct requests to the backend, as most reads and initial writes happen locally.

2. When to Use Offline-First?

Offline-first is particularly beneficial for apps where:

- **Data creation/modification is frequent:** Chat apps, note-taking apps, to-do lists, expense trackers.
- **Users operate in varied network conditions:** Field service apps, travel apps,

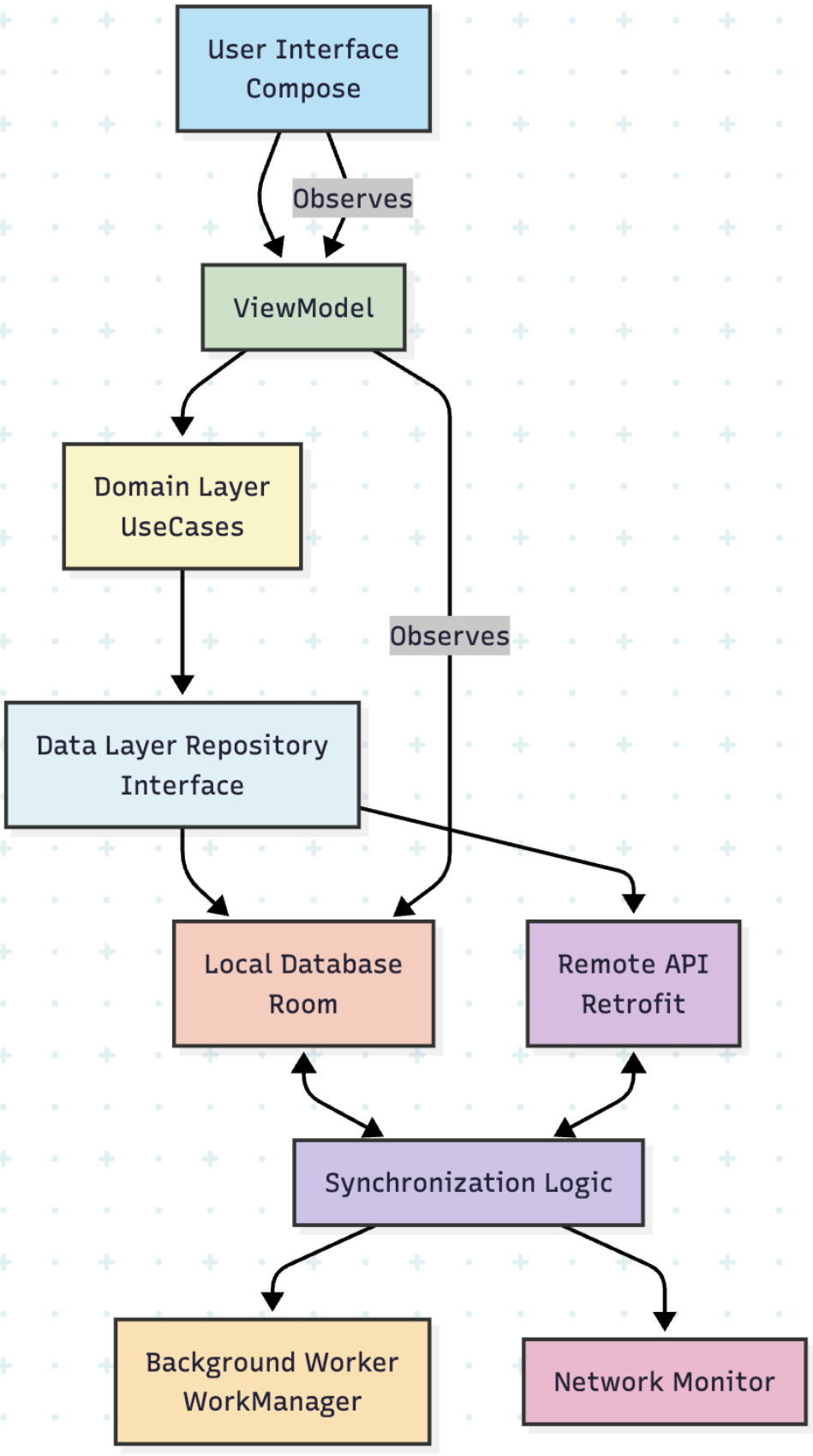
delivery apps.
- **High performance and responsiveness are critical:** Any app where users expect instant feedback.
- **Data needs to be accessible for extended periods offline:** E-readers, offline maps, content consumption apps.
- **Data integrity is important even with intermittent connectivity:** Financial apps, inventory management.

It might be overkill for apps that are purely read-only and require real-time, always-up-to-date information (e.g., live stock tickers, real-time sports scores where historical data isn't the primary focus).

3. Core Principles & Architecture Types

The fundamental principle of offline-first is that the **local database is the primary source of truth for the UI.** The remote backend is a secondary source for synchronization.

Conceptual Architecture Diagram:


User Interface
Compose
Observes
ViewModel
Domain Layer
UseCases
Observes
Data Layer Repository
Interface
Local Database
Room
Remote API
Retrofit
Synchronization Logic
Background Worker
WorkManager
Network Monitor

Types of Offline-First Architectures (Approaches):

1. **Read-Through Cache:** Simplest. Data is fetched from the network and cached locally. Subsequent reads try local first, then network. Writes always go to the network. Limited offline functionality.
2. **Write-Through Cache:** Writes go to both local cache and network simultaneously. If network fails, local write succeeds, and network write is retried. Better, but still relies on an immediate network for writes.
3. **Local-First (True Offline-First):**
    - **All reads and writes happen against the local database.**
    - User actions are immediately reflected in the UI (optimistic updates).
    - A separate synchronization mechanism handles pushing local changes to the remote and pulling remote changes to the local database.

4. How to Integrate Offline-First into an Existing Mobile App

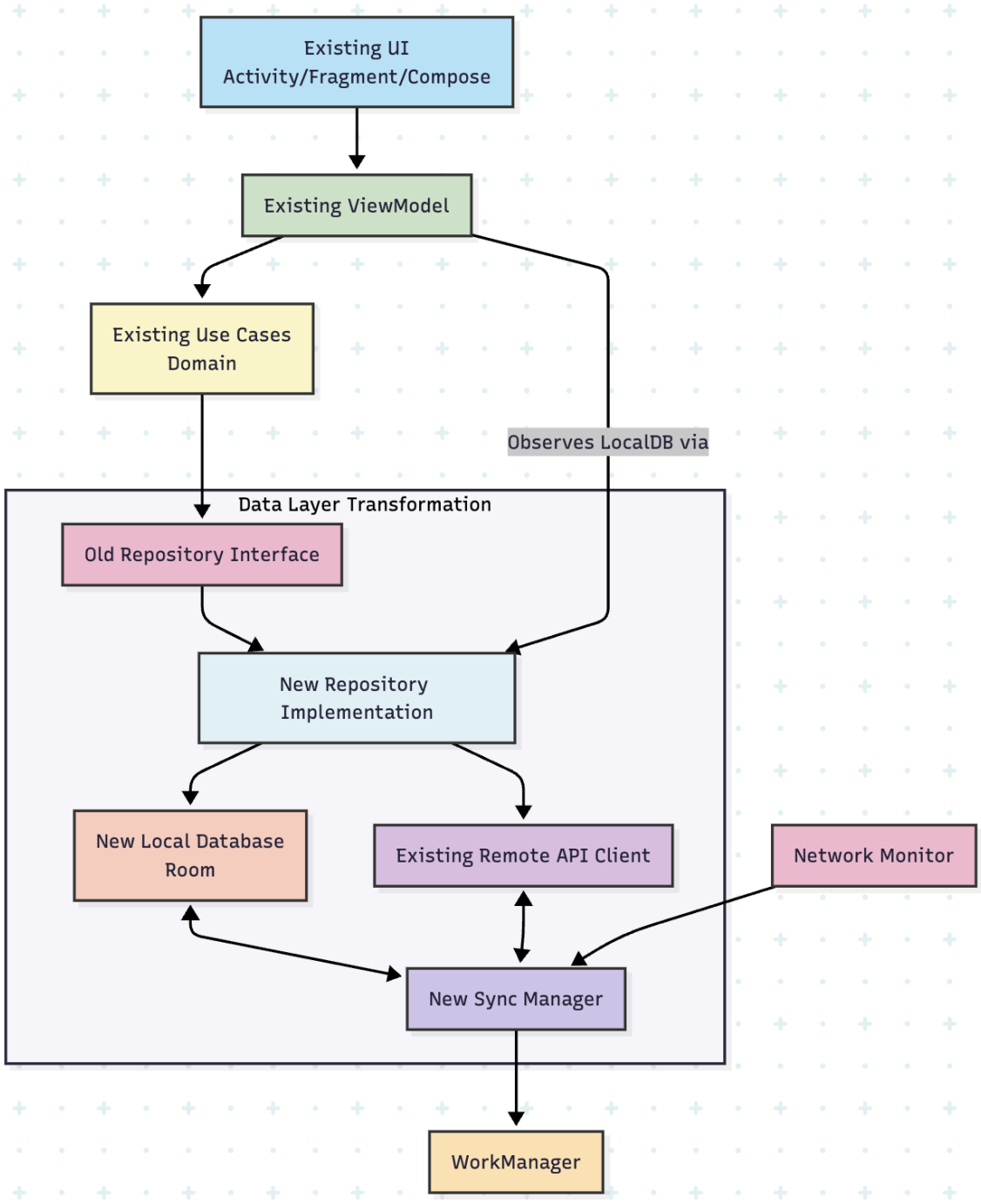
Integrating offline-first into an existing app primarily involves modifying the **Data Layer** and introducing a dedicated **Synchronization Layer**.

Steps:

1. **Establish Local Database as Source of Truth:**
    - Introduce a robust local database (e.g., Room for Android, Realm, SQLite).
    - All data displayed to the UI *must* come from this local database.
    - The Repository implementation (in the Data Layer) will now primarily read from and write to this local database.
2. **Decouple Remote API Interaction:**
    - The Repository will still have a dependency on a `RemoteDataSource` (e.g., Retrofit service).
    - However, the Repository's save methods will *not* directly call the remote API. Instead, they will save to the local database and mark the data as "pending sync" or "dirty."
3. **Implement Synchronization Logic:**
    - Create a dedicated `SyncManager` or `SyncService` component.
    - This component is responsible for:
        - Detecting network availability.
        - Identifying local changes that need to be pushed to the remote.
        - Fetching remote changes and applying them to the local database.
        - Handling conflicts (if any).
        - Managing queues for outgoing data.

4. **Integrate Background Sync:**
    - Use Android's `WorkManager` for reliable, battery-efficient background synchronization.
    - `WorkManager` can schedule tasks based on network availability, device idle state, and periodic intervals.
5. **Update UI to Observe Local Changes:**
    - Ensure your `ViewModels` (or Presenters) collect Flows (or other observable streams) directly from the local database via the Repository. This ensures the UI is always up-to-date with the local source of truth.

Diagram: Integrating into Existing App Layers



5. Data Synchronization Mechanisms

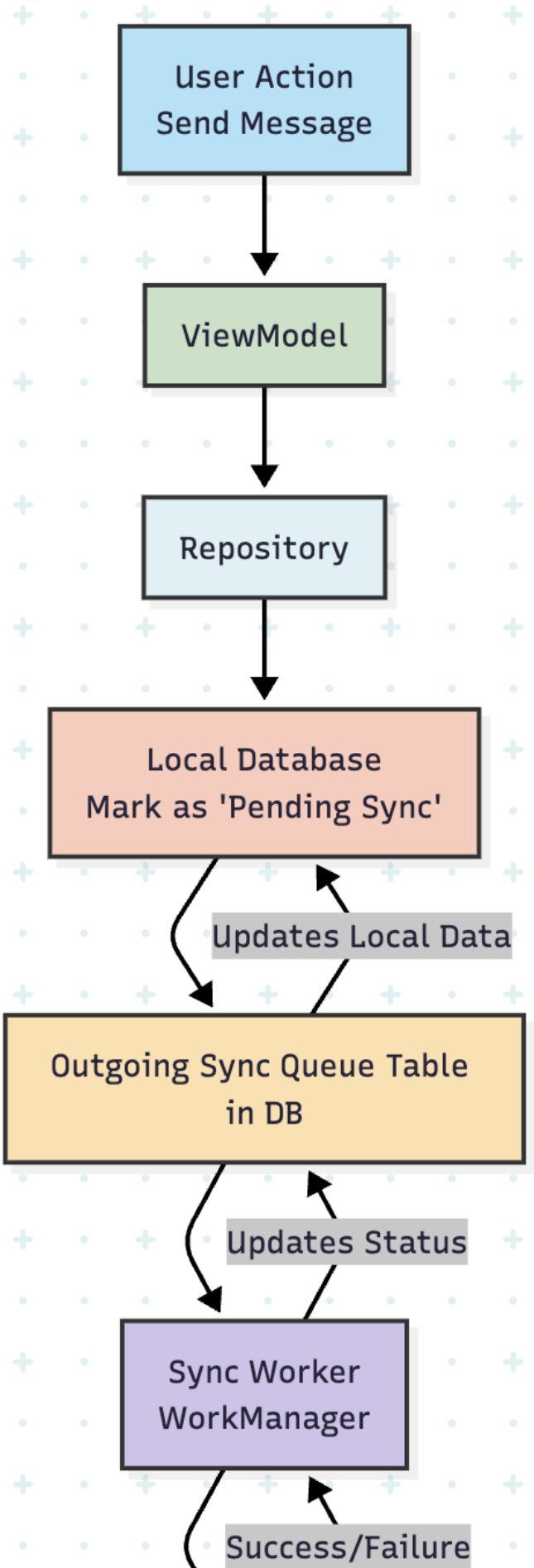
The heart of offline-first is robust data synchronization.

5.1. Queuing Data for Sync

When a user performs an action offline (e.g., sends a message), the data needs to be stored locally and then reliably sent to the server when connectivity is restored. This is where queuing comes in.

- **Why Queue?**
  - **Reliability:** Ensures data isn't lost if the app crashes or network drops before successful sync.
  - **Order Preservation:** Maintains the order of operations (e.g., message A sent before message B).
  - **Retry Logic:** Automatically retries failed sync attempts.
  - **Decoupling:** Separates the UI action from the network operation.
- **Types of Queuing:**
  1. **In-Memory Queue:** Simplest. A `MutableList` or `Channel` in a `ViewModel` or `Service`. Data is lost if the app/process dies. Only for very short-lived or non-critical operations.
  2. **Persistent Queue (Recommended for Offline-First):**
     - **Local Database (Room):** The most common and robust approach. Store "sync operations" or "dirty data" directly in your local database. Each operation (e.g., "send message", "update profile") gets an entry in a dedicated "sync queue" table, often with a status (pending, syncing, failed) and retry_count.
     - **WorkManager:** Can act as a persistent queue orchestrator. You enqueue `WorkRequests` (e.g., `OneTimeWorkRequest` for each outgoing message) that are persisted by `WorkManager` and executed when constraints (like network) are met.

Diagram: Data Queuing for Outgoing Sync


User Action
Send Message
ViewModel
Repository
Local Database
Mark as 'Pending Sync'
Updates Local Data
Outgoing Sync Queue Table
in DB
Updates Status
Sync Worker
WorkManager
Success/Failure

5.2. Keeping the Local Database Updated (Incoming Sync)

- **Polling:** Periodically fetch all or new data from the server and update the local database. Simple, but can be inefficient.
- **WebSockets/Push Notifications:** For real-time updates (like chat messages), a WebSocket connection can push changes from the server directly to the client, which then updates the local database. Push notifications (FCM) can wake up the app to trigger a sync for less critical updates.
- **Change Detection:** The server can provide a timestamp or version number. The client requests only data changed since its last sync.
- **Reconciliation & Conflict Resolution:**
  - When both local and remote data have changed, a conflict arises.
  - **Last-Write-Wins:** Simplest. The most recent change (based on timestamp) wins.
  - **Client-Wins/Server-Wins:** One side always takes precedence.
  - **Merge:** Attempt to combine changes (complex, requires domain knowledge).
  - **User Intervention:** Prompt the user to resolve conflicts (rarely desired for good UX).

6. Best Practices for Offline-First

- **Local Database as Primary Source:** Always read from local DB for UI updates.
- **Optimistic UI Updates:** Reflect user actions immediately in the UI (by saving to local DB), even before server confirmation.
- **Clear Sync Status Feedback:** Inform the user about pending syncs, successful syncs, and sync errors (e.g., "Sending...", "Sent", "Failed to send").
- **Robust Error Handling & Retries:** Implement exponential backoff for network retries.
- **Data Modeling for Offline:** Design your local database schema to efficiently store all necessary data, including sync metadata (e.g., `isSynced: Boolean`, `lastModified: Long`, `syncStatus: String`).
- **Idempotent Operations:** Design your API endpoints and sync logic so that performing an operation multiple times has the same effect as performing it once. This is crucial for retries.
- **Network Monitoring:** Actively monitor network connectivity to trigger syncs when online.
- **Background Processing:** Leverage `WorkManager` for reliable background syncs.
- **Security:** Encrypt sensitive data in the local database. Be mindful of what data is stored offline.

7. Example App: Offline-First Chat App

Let's build a simplified "Offline-First Chat App" to demonstrate these concepts.

Use Case:

- Users can send messages.
- Messages appear instantly in the sender's UI.
- If offline, messages are queued and sent when online.
- Messages from other users appear when online (incoming sync).

Architecture Overview:

- **Domain Layer:** `Message` (data model), `ChatRepository` (interface).
- **Data Layer:**
  - `MessageEntity` (Room entity for local storage).
  - `MessageDao` (Room DAO for DB operations).
  - `LocalChatDataSource` (interacts with Room).
  - `RemoteChatDataSource` (simulates API calls).
  - `ChatRepositoryImpl` (implements `ChatRepository`, orchestrates local/remote, marks messages for sync).
- **Synchronization Layer:**
  - `SyncWorker` (WorkManager worker for background sync).
  - `SyncManager` (initiates WorkManager, monitors network).
- **Presentation Layer:**
  - `ChatUiState` (UI state for chat screen).
  - `ChatViewModel` (manages ChatUiState, handles UI events, triggers sync).
  - `ChatScreen` (Compose UI).

Code Implementation (Step-by-Step)

Step 7.1: Domain Layer (Models & Repository Interface)

These are the pure Kotlin definitions, independent of Android or specific data sources.

File: domain/Message.kt

```
package com.yourappname.chatapp.domain

import java.util.Date
import java.util.UUID

// Represents a single chat message in the domain
```

```
data class Message(
    val id: String = UUID.randomUUID().toString(), // Unique ID for the message
    val senderId: String,          // ID of the user who sent the message
    val text: String,              // The content of the message
    val timestamp: Date = Date(),  // When the message was created/sent
    val isSentByMe: Boolean,       // True if the current user sent this message
    val status: MessageStatus = MessageStatus.SENT_OR_PENDING // Current sync
status
)

enum class MessageStatus {
    SENT_OR_PENDING, // Message created locally, waiting to be sent to server
    SENT_TO_SERVER,  // Message successfully sent to server
    FAILED_TO_SEND   // Message failed to send to server
}
```

Explanation: We've added `senderId`, `isSentByMe`, and `status` to track message ownership and sync state.

File: domain/ChatRepository.kt

```
package com.yourappname.chatapp.domain

import kotlinx.coroutines.flow.Flow

// Defines the contract for chat message operations
interface ChatRepository {

    /**
     * Send a new message. This will typically save locally first, then queue for
sync.
     * @param message The message to send.
     */
    suspend fun sendMessage(message: Message)

    /**
     * Gets a flow of all messages, ordered by timestamp (ascending).
     * The UI will observe this flow for real-time updates from the local database.
     * @return A Flow emitting lists of Messages.
    */
    fun getMessages(): Flow<List<Message>>

    /**
     * Fetches messages from the remote server and updates the local database.
     * This is part of the incoming sync process.
     */
```

```
    suspend fun fetchAndSyncRemoteMessages()

    /**
     * Retrieves messages that are locally created but not yet sent to the server.
     * Used by the sync worker to identify messages to push.
     * @return A list of messages pending synchronization.
     */
    suspend fun getPendingOutgoingMessages(): List<Message>

    /**
     * Updates the status of a message (e.g., after successful send).
     * @param messageId The ID of the message to update.
     * @param newStatus The new status to set.
     */
    suspend fun updateMessageStatus(messageId: String, newStatus: MessageStatus)
}
```

Explanation:

- `sendMessage`: The primary way the UI sends a message. It will internally handle local saving and queuing.
- `getMessages()`: Provides a Flow from the local database, ensuring the UI is always up-to-date.
- `fetchAndSyncRemoteMessages()`: This is the core method for pulling messages from the server.
- `getPendingOutgoingMessages()` and `updateMessageStatus()`: These are specific methods needed by our sync logic.

Step 7.2: Data Layer - Local Database (Room)

We'll use Room Persistence Library for our local database.

File: data/local/MessageEntity.kt

```
package com.yourappname.chatapp.data.local

import androidx.room.Entity
import androidx.room.PrimaryKey
import com.yourappname.chatapp.domain.MessageStatus
import java.util.Date

// Room Entity representing a message in the local database
@Entity(tableName = "messages")
data class MessageEntity(
```

```
    @PrimaryKey val id: String,
    val senderId: String,
    val text: String,
    val timestamp: Date,
    val isSentByMe: Boolean,
    val status: MessageStatus // Sync status for this message
) {
    // Helper function to convert Entity to Domain Model
    fun toDomainModel(): com.yourappname.chatapp.domain.Message {
        return com.yourappname.chatapp.domain.Message(
            id = id,
            senderId = senderId,
            text = text,
            timestamp = timestamp,
            isSentByMe = isSentByMe,
            status = status
        )
    }
}

// Helper function to convert Domain Model to Entity
fun com.yourappname.chatapp.domain.Message.toEntity(): MessageEntity {
    return MessageEntity(
        id = id,
        senderId = senderId,
        text = text,
        timestamp = timestamp,
        isSentByMe = isSentByMe,
        status = status
    )
}
```

Explanation: `MessageEntity` is a Room-specific representation. It mirrors `Message` but includes Room annotations. We add helper functions for easy conversion between domain and entity models.

File: data/local/MessageDao.kt

```
package com.yourappname.chatapp.data.local

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Update
```

```
import kotlinx.coroutines.flow.Flow

// Data Access Object for MessageEntity
@Dao
interface MessageDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE) // Replace if message with
same ID exists
    suspend fun insertMessage(message: MessageEntity)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertMessages(messages: List<MessageEntity>)

    @Query("SELECT * FROM messages ORDER BY timestamp ASC")
    fun getMessages(): Flow<List<MessageEntity>> // Returns a Flow of entities

    @Query("SELECT * FROM messages WHERE id = :messageId")
    suspend fun getMessageById(messageId: String): MessageEntity?

    @Update
    suspend fun updateMessage(message: MessageEntity)

    @Query("SELECT * FROM messages WHERE isSentByMe = 1 AND status = :status ORDER
BY timestamp ASC")
    suspend fun getOutgoingMessagesByStatus(status: MessageStatus):
List<MessageEntity>

    @Query("DELETE FROM messages WHERE id = :messageId")
    suspend fun deleteMessage(messageId: String)
}
```

Explanation: The `DAO` defines the database operations. `getMessages()` returns a `Flow<List<MessageEntity>>` which is crucial for observing local data changes.

File: data/local/DateConverter.kt (for Room to store Dates)

```
package com.yourappname.chatapp.data.local

import androidx.room.TypeConverter
import java.util.Date

// Type converter for Room to store and retrieve Date objects
class DateConverter {
    @TypeConverter
    fun fromTimestamp(value: Long?): Date? {
```

```
        return value?.let { Date(it) }
    }

    @TypeConverter
    fun dateToTimestamp(date: Date?): Long? {
        return date?.time
    }
}
```

File: data/local/MessageStatusConverter.kt (for Room to store Enum)

```
package com.yourappname.chatapp.data.local

import androidx.room.TypeConverter
import com.yourappname.chatapp.domain.MessageStatus

// Type converter for Room to store and retrieve MessageStatus enum
class MessageStatusConverter {
    @TypeConverter
    fun fromMessageStatus(value: MessageStatus): String {
        return value.name
    }

    @TypeConverter
    fun toMessageStatus(value: String): MessageStatus {
        return MessageStatus.valueOf(value)
    }
}
```

File: data/local/AppDatabase.kt

```
package com.yourappname.chatapp.data.local

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters

// Room Database definition
@Database(entities = [MessageEntity::class], version = 1, exportSchema = false)
@TypeConverters(DateConverter::class, MessageStatusConverter::class) // Register
type converters
abstract class AppDatabase : RoomDatabase() {
```

```
    abstract fun messageDao(): MessageDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "chat_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Step 7.3: Data Layer - Remote API (Simulated)

We'll simulate a backend API for sending and receiving messages.

File: data/remote/ChatApiService.kt

```
package com.yourappname.chatapp.data.remote

import com.yourappname.chatapp.domain.Message
import kotlinx.coroutines.delay
import java.util.Date
import java.util.UUID

// Data Transfer Object (DTO) for messages sent to/received from API
data class MessageDto(
    val id: String,
    val senderId: String,
    val text: String,
    val timestamp: Long // Use Long for timestamp in DTOs
) {
    fun toDomainModel(): Message {
        return Message(
            id = id,
            senderId = senderId,
            text = text,
```

```
            timestamp = Date(timestamp),
            isSentByMe = false, // This will be set correctly by the
repository/sync logic
            status = com.yourappname.chatapp.domain.MessageStatus.SENT_TO_SERVER
        )
    }
}

// Simulated Remote Chat API Service
class ChatApiService {

    // Simulate messages from other users
    private val remoteMessages = mutableListOf(
        MessageDto("remote-1", "user2", "Hey there!", Date().time - 60000),
        MessageDto("remote-2", "user3", "How are you?", Date().time - 30000)
    )

    /**
     * Simulates sending a message to the remote server.
     * @param messageDto The message to send.
     * @return True if sent successfully, false otherwise.
     */
    suspend fun sendMessage(messageDto: MessageDto): Boolean {
        println("RemoteAPI: Sending message: ${messageDto.text}")
        delay(1000) // Simulate network delay
        val success = (0..10).random() > 1 // 80% success rate
        if (success) {
            println("RemoteAPI: Message sent successfully: ${messageDto.text}")
        } else {
            println("RemoteAPI: Failed to send message: ${messageDto.text}")
        }
        return success
    }

    /**
     * Simulates fetching new messages from the remote server.
     * In a real app, this would query messages after a certain timestamp.
     * @return A list of new messages.
     */
    suspend fun getNewMessages(): List<MessageDto> {
        println("RemoteAPI: Fetching new messages...")
        delay(1500) // Simulate network delay

        // Simulate new messages appearing over time
        val newMessages = mutableListOf<MessageDto>()
        if (System.currentTimeMillis() % 2 == 0) { // Randomly add new messages
            newMessages.add(MessageDto(UUID.randomUUID().toString(),
```

```
"user${(4..5).random()}", "New message from user ${(4..5).random()}", Date().time))
        }
        if (System.currentTimeMillis() % 3 == 0) {
            newMessages.add(MessageDto(UUID.randomUUID().toString(),
"user${(6..7).random()}", "Another new message from user ${(6..7).random()}",
Date().time + 1000))
        }
        println("RemoteAPI: Fetched ${newMessages.size} new messages.")
        return newMessages + remoteMessages // Return existing + new
    }
}
```

Explanation: `MessageDto` is a simple DTO. `ChatApiService` has `sendMessage` (simulates outgoing) and `getNewMessages` (simulates incoming). It includes random success/failure for `sendMessage` to test retry logic.

Step 7.4: Data Layer - Repository Implementation (Orchestrates Local & Remote)

This is the core of our offline-first data layer. It implements the `ChatRepository` interface and decides how to interact with local and remote data sources.

File: data/ChatRepositoryImpl.kt

```
package com.yourappname.chatapp.data

import com.yourappname.chatapp.data.local.AppDatabase
import com.yourappname.chatapp.data.local.toDomainModel
import com.yourappname.chatapp.data.local.toEntity
import com.yourappname.chatapp.data.remote.ChatApiService
import com.yourappname.chatapp.domain.Message
import com.yourappname.chatapp.domain.MessageStatus
import com.yourappname.chatapp.domain.ChatRepository
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.withContext
import kotlinx.coroutines.Dispatchers

// Implementation of ChatRepository that orchestrates local and remote data
class ChatRepositoryImpl(
    private val database: AppDatabase,
    private val chatApiService: ChatApiService,
    private val currentUserId: String // Assume current user ID is provided
) : ChatRepository {

    private val messageDao = database.messageDao()
```

```
    override suspend fun sendMessage(message: Message) {
        // 1. Save message to local database immediately (optimistic update)
        // Mark it as PENDING_SYNC
        val messageToSave = message.copy(
            isSentByMe = true, // Ensure this is true for outgoing messages
            status = MessageStatus.SENT_OR_PENDING
        )
        messageDao.insertMessage(messageToSave.toEntity())
        println("Repository: Message saved locally as PENDING:
${messageToSave.text}")

        // 2. The sync worker will pick this up later
    }

    override fun getMessages(): Flow<List<Message>> {
        // Always read from the local database for the UI
        return messageDao.getMessages().map { entities ->
            entities.map { it.toDomainModel() } // Convert entities to domain
models
        }
    }

    override suspend fun fetchAndSyncRemoteMessages() {
        println("Repository: Starting incoming sync from remote...")
        try {
            val remoteMessagesDto = chatApiService.getNewMessages()
            val incomingMessages = remoteMessagesDto.map { dto ->
                // Convert DTO to domain model, ensuring status is SENT_TO_SERVER
                dto.toDomainModel().copy(
                    isSentByMe = (dto.senderId == currentUserId), // Correctly set
based on senderId
                    status = MessageStatus.SENT_TO_SERVER // These messages are
already synced
                )
            }
            // Insert or update incoming messages into the local database
            // OnConflictStrategy.REPLACE in DAO handles updates for existing IDs
            messageDao.insertMessages(incomingMessages.map { it.toEntity() })
            println("Repository: Incoming sync complete. Inserted/updated
${incomingMessages.size} messages locally.")
        } catch (e: Exception) {
            println("Repository: Failed to fetch remote messages: ${e.message}")
            throw e // Re-throw to be caught by sync worker for retry logic
        }
    }
```

```
    override suspend fun getPendingOutgoingMessages(): List<Message> {
        // Get messages that are locally created and waiting to be sent
        return
messageDao.getOutgoingMessagesByStatus(MessageStatus.SENT_OR_PENDING)
            .map { it.toDomainModel() }
    }

    override suspend fun updateMessageStatus(messageId: String, newStatus:
MessageStatus) {
        // Get the current message, update its status, and save back to DB
        val messageEntity = messageDao.getMessageById(messageId)
        if (messageEntity != null) {
            val updatedEntity = messageEntity.copy(status = newStatus)
            messageDao.updateMessage(updatedEntity)
            println("Repository: Message ID $messageId status updated to
$newStatus")
        }
    }
}
```

Explanation:

- `sendMessage`: Immediately saves the message to Room with `SENT_OR_PENDING` status. The UI will see this instantly.
- `getMessages`: Always reads from `messageDao.getMessages()`, which returns a `Flow`. This ensures the UI reflects local changes immediately.
- `fetchAndSyncRemoteMessages`: Calls the `chatApiService` to get new messages, converts them, and inserts/updates them into the local Room DB.
- `getPendingOutgoingMessages`: Used by the sync worker to find messages that need to be pushed.
- `updateMessageStatus`: Changes a message's status in the local DB after sync attempts.

Step 7.5: Synchronization Layer (WorkManager & SyncManager)

This layer handles the background synchronization logic.

File: data/sync/SyncWorker.kt

```
package com.yourappname.chatapp.data.sync

import android.content.Context
import androidx.work.CoroutineWorker
```

```
import androidx.work.WorkerParameters
import com.yourappname.chatapp.data.ChatRepositoryImpl
import com.yourappname.chatapp.data.local.AppDatabase
import com.yourappname.chatapp.data.remote.ChatApiService
import com.yourappname.chatapp.domain.MessageStatus
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext

// Worker responsible for performing data synchronization
class SyncWorker(
    appContext: Context,
    workerParams: WorkerParameters
) : CoroutineWorker(appContext, workerParams) {

    // Dependencies would typically be injected via a custom WorkerFactory in a
real app
    private val chatRepository: ChatRepositoryImpl by lazy {
        ChatRepositoryImpl(
            AppDatabase.getDatabase(appContext),
            ChatApiService(),
            "myUserId" // Replace with actual current user ID
        )
    }

    override suspend fun doWork(): Result = withContext(Dispatchers.IO) {
        println("SyncWorker: Starting sync operation...")
        try {
            // --- Outgoing Sync (Push local changes to remote) ---
            val pendingMessages = chatRepository.getPendingOutgoingMessages()
            if (pendingMessages.isNotEmpty()) {
                println("SyncWorker: Found ${pendingMessages.size} pending outgoing
messages.")
                for (message in pendingMessages) {
                    try {
                        val success = chatRepository.chatApiService.sendMessage(
                            com.yourappname.chatapp.data.remote.MessageDto(
                                id = message.id,
                                senderId = message.senderId,
                                text = message.text,
                                timestamp = message.timestamp.time
                            )
                        )
                        if (success) {
                            chatRepository.updateMessageStatus(message.id,
MessageStatus.SENT_TO_SERVER)
                            println("SyncWorker: Successfully sent message ID:
${message.id}")
```

```
                        } else {
                            // Mark as failed for now, retry logic handled by
WorkManager if transient
                            chatRepository.updateMessageStatus(message.id,
MessageStatus.FAILED_TO_SEND)
                            println("SyncWorker: Failed to send message ID:
${message.id}")
                        }
                    } catch (e: Exception) {
                        println("SyncWorker: Exception sending message ID
${message.id}: ${e.message}")
                        // Mark as failed, WorkManager will retry this worker if
it's a transient error
                        chatRepository.updateMessageStatus(message.id,
MessageStatus.FAILED_TO_SEND)
                    }
                }
            } else {
                println("SyncWorker: No pending outgoing messages.")
            }

            // --- Incoming Sync (Pull remote changes to local) ---
            chatRepository.fetchAndSyncRemoteMessages()
            println("SyncWorker: Incoming sync completed.")

            Result.success() // Sync successful
        } catch (e: Exception) {
            println("SyncWorker: Sync operation failed: ${e.message}")
            Result.retry() // Retry if failed due to transient issues (e.g.,
network)
        }
    }
}
```

Explanation:

- `CoroutineWorker` allows suspend functions.
- `doWork()` contains the actual sync logic:
  - It fetches `getPendingOutgoingMessages()` from the repository.
  - For each pending message, it tries to `sendMessage` via the `chatApiService`.
  - It updates the message's status in the local DB (`SENT_TO_SERVER` or `FAILED_TO_SEND`).
  - It then calls `fetchAndSyncRemoteMessages()` to pull new messages

from the server.

- `Result.success()` indicates completion, `Result.retry()` indicates a transient failure that `WorkManager` should retry.

File: data/sync/SyncManager.kt

```
package com.yourappname.chatapp.data.sync

import android.content.Context
import androidx.work.Constraints
import androidx.work.ExistingPeriodicWorkPolicy
import androidx.work.NetworkType
import androidx.work.PeriodicWorkRequestBuilder
import androidx.work.WorkManager
import java.util.concurrent.TimeUnit

// Manages the scheduling and triggering of synchronization
object SyncManager {

    private const val SYNC_WORK_NAME = "chat_sync_work"

    /**
     * Schedules periodic background synchronization using WorkManager.
     * This ensures the app syncs even when not actively in use.
     * @param context Application context.
     */
    fun schedulePeriodicSync(context: Context) {
        val constraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED) // Only sync when
connected
            .setRequiresBatteryNotLow(true) // Don't sync if battery is low
            .build()

        val syncRequest = PeriodicWorkRequestBuilder<SyncWorker>(
            repeatInterval = 15, // Sync every 15 minutes
            repeatIntervalTimeUnit = TimeUnit.MINUTES
        )
            .setConstraints(constraints)
            .build()

        WorkManager.getInstance(context).enqueueUniquePeriodicWork(
            SYNC_WORK_NAME,
            ExistingPeriodicWorkPolicy.KEEP, // If work already exists, keep the
existing one
            syncRequest
        )
```

```
        println("SyncManager: Periodic sync scheduled.")
    }

    /**
     * Triggers an immediate one-time sync. Useful after a user action (e.g.,
sending a message)
     * or when the app comes to foreground.
     * @param context Application context.
     */
    fun triggerImmediateSync(context: Context) {
        val constraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .build()

        val syncRequest = androidx.work.OneTimeWorkRequestBuilder<SyncWorker>()
            .setConstraints(constraints)
            .build()

        WorkManager.getInstance(context).enqueue(syncRequest)
        println("SyncManager: Immediate sync triggered.")
    }

    // You might also want a method to cancel sync if needed
    fun cancelPeriodicSync(context: Context) {
        WorkManager.getInstance(context).cancelUniqueWork(SYNC_WORK_NAME)
        println("SyncManager: Periodic sync cancelled.")
    }
}
```

Explanation:

- `schedulePeriodicSync`: Sets up a recurring `SyncWorker` task that runs every 15 minutes when the device is connected to the network and battery is not low. KEEP policy ensures only one such work is active.
- `triggerImmediateSync`: Enqueues a single `SyncWorker` task that runs as soon as network constraints are met. This is ideal for pushing messages immediately after a user sends them.

Step 7.6: Presentation Layer (UI State, ViewModel, Compose UI)

Finally, we connect everything to the UI.

File: presentation/chat/ChatUiState.kt

```
package com.yourappname.chatapp.presentation.chat
```

```
import com.yourappname.chatapp.domain.Message

// Represents the entire UI state of the chat screen
data class ChatUiState(
    val messages: List<Message> = emptyList(), // The list of messages to display
    val currentInput: String = "",           // Text in the message input field
    val isLoading: Boolean = false,          // Whether initial messages are
loading
    val error: String? = null,               // Any error message to show
    val isSending: Boolean = false,          // Whether a message is currently
being sent
    val syncStatus: String = "Idle"          // Display sync status to user (e.g.,
"Syncing...", "Last synced: X min ago")
)
```

File: presentation/chat/ChatViewModel.kt

```
package com.yourappname.chatapp.presentation.chat

import android.content.Context
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.work.WorkInfo
import androidx.work.WorkManager
import com.yourappname.chatapp.data.sync.SyncWorker
import com.yourappname.chatapp.domain.ChatRepository
import com.yourappname.chatapp.domain.Message
import com.yourappname.chatapp.data.sync.SyncManager // For triggering sync
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.catch
import kotlinx.coroutines.flow.onStart
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.asSharedFlow
import java.util.Date

class ChatViewModel(
    private val chatRepository: ChatRepository,
    private val applicationContext: Context, // Need context for WorkManager
    private val currentUserId: String = "myUserId" // Assume current user ID
) : ViewModel() {
```

```
    // --- UI State ---
    private val _uiState = MutableStateFlow(ChatUiState(isLoading = true))
    val uiState: StateFlow<ChatUiState> = _uiState.asStateFlow()

    // --- Side Effects ---
    private val _sideEffects = MutableSharedFlow<ChatSideEffect>()
    val sideEffects = _sideEffects.asSharedFlow()

    sealed class ChatSideEffect {
        data class ShowSnackbar(val message: String) : ChatSideEffect()
        // Add navigation side effects if chat screen navigates
    }

    init {
        // 1. Collect messages from the local repository for UI updates
        viewModelScope.launch {
            chatRepository.getMessages()
                .onStart { _uiState.update { it.copy(isLoading = true, error =
null) } }
                .catch { e -> _uiState.update { it.copy(isLoading = false, error =
"Failed to load messages: ${e.message}") } }
                .collect { messages ->
                    _uiState.update { it.copy(isLoading = false, messages =
messages, error = null) }
                }
        }

        // 2. Observe WorkManager status to update syncStatus in UI
        viewModelScope.launch {
            WorkManager.getInstance(applicationContext)
                .getWorkInfosForUniqueWorkLiveData("chat_sync_work") // Observe the
periodic sync
                .asFlow() // Convert LiveData to Flow
                .collect { workInfos ->
                    val syncInfo = workInfos.firstOrNull() // Get info for our sync
work
                    val statusText = when (syncInfo?.state) {
                        WorkInfo.State.ENQUEUED -> "Queued for sync"
                        WorkInfo.State.RUNNING -> "Syncing..."
                        WorkInfo.State.SUCCEEDED -> "Last synced: ${Date()}" //
Simplified
                        WorkInfo.State.FAILED -> "Sync failed"
                        WorkInfo.State.BLOCKED -> "Sync blocked"
                        WorkInfo.State.CANCELLED -> "Sync cancelled"
                        null -> "No sync scheduled"
                    }
                    _uiState.update { it.copy(syncStatus = statusText) }
```

```
                }
            }

            // 3. Trigger an initial immediate sync when ViewModel is created
            SyncManager.triggerImmediateSync(applicationContext)
            SyncManager.schedulePeriodicSync(applicationContext) // Schedule periodic
sync
        }

        // --- Event Handlers (UI Events) ---

        fun onInputChanged(newInput: String) {
            _uiState.update { it.copy(currentInput = newInput) }
        }

        fun onSendClicked() {
            val messageText = _uiState.value.currentInput.trim()
            if (messageText.isBlank()) {
                viewModelScope.launch {
_sideEffects.emit(ChatSideEffect.ShowSnackbar("Message cannot be empty")) }
                return
            }

            _uiState.update { it.copy(isSending = true, currentInput = "") } // Clear
input and show sending status

            viewModelScope.launch {
                try {
                    val newMessage = Message(
                        senderId = currentUserId,
                        text = messageText,
                        isSentByMe = true,
                        timestamp = Date(),
                        status = MessageStatus.SENT_OR_PENDING // Will be updated by
sync worker
                    )
                    chatRepository.sendMessage(newMessage) // Save locally
                    // The sync worker will automatically pick this up due to
triggerImmediateSync() below

                    _uiState.update { it.copy(isSending = false) }
                    SyncManager.triggerImmediateSync(applicationContext) // Trigger
immediate sync
                } catch (e: Exception) {
                    _uiState.update { it.copy(isSending = false) }
                    _sideEffects.emit(ChatSideEffect.ShowSnackbar("Failed to send
message locally: ${e.message}"))
```

```
            }
        }
    }

    // Add other event handlers like onRetrySendClicked, etc.
}

// presentation/chat/ChatViewModelFactory.kt (Manual Factory)
import android.content.Context
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import com.yourappname.chatapp.data.ChatRepositoryImpl
import com.yourappname.chatapp.data.local.AppDatabase
import com.yourappname.chatapp.data.remote.ChatApiService

class ChatViewModelFactory(private val context: Context) :
ViewModelProvider.Factory {
    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(ChatViewModel::class.java)) {
            val database = AppDatabase.getDatabase(context.applicationContext)
            val chatApiService = ChatApiService()
            val chatRepository = ChatRepositoryImpl(database, chatApiService,
"myUserId") // Pass current user ID
            return ChatViewModel(chatRepository, context.applicationContext,
"myUserId") as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Explanation:

- `ChatViewModel` collects messages from `chatRepository.getMessages()` (which comes from `Room`) to keep `_uiState.messages` updated.
- It also observes `WorkManager's` status to update `_uiState.syncStatus`.
- `onSendClicked` immediately saves the message locally, updates the UI, and then `triggerImmediateSync` to try sending it.
- `_sideEffects: SharedFlow` is used for `ShowSnackbar` messages.

File: presentation/chat/ChatScreen.kt

```
package com.yourappname.chatapp.presentation.chat
```

```
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.*
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Send
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import androidx.lifecycle.viewmodel.compose.viewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import com.yourappname.chatapp.domain.Message
import com.yourappname.chatapp.domain.MessageStatus
import java.text.SimpleDateFormat
import java.util.Locale

@Composable
fun ChatScreen(
    viewModel: ChatViewModel = viewModel(), // Get ViewModel
    snackbarHostState: SnackbarHostState, // From parent Scaffold
    currentUserId: String = "myUserId" // Pass current user ID for UI logic
) {
    // --- Collect UI State ---
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState() // For scrolling to bottom

    // --- Collect Side Effects ---
    LaunchedEffect(Unit) {
        viewModel.sideEffects.collect { effect ->
            when (effect) {
                is ChatViewModel.ChatSideEffect.ShowSnackbar -> {
                    snackbarHostState.showSnackbar(effect.message)
                }
            }
        }
    }

    // --- Auto-scroll to bottom when new messages arrive ---
    LaunchedEffect(uiState.messages.size) {
        if (uiState.messages.isNotEmpty()) {
            listState.animateScrollToItem(uiState.messages.lastIndex)
        }
```

```
    }

    Scaffold(
        topBar = {
            TopAppBar(title = { Text("Offline-First Chat") })
        },
        snackbarHost = { SnackbarHost(snackbarHostState) }
    ) { paddingValues ->
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(paddingValues)
        ) {
            // --- Messages List ---
            Box(
                modifier = Modifier
                    .weight(1f)
                    .fillMaxWidth(),
                contentAlignment = Alignment.Center // For loading/empty state
            ) {
                when {
                    uiState.isLoading -> CircularProgressIndicator()
                    uiState.error != null -> Text("Error: ${uiState.error}", color
= MaterialTheme.colors.error)
                    uiState.messages.isEmpty() -> Text("Start a conversation!")
                    else -> {
                        LazyColumn(
                            state = listState,
                            modifier = Modifier.fillMaxSize(),
                            contentPadding = PaddingValues(8.dp),
                            verticalArrangement = Arrangement.spacedBy(8.dp)
                        ) {
                            items(uiState.messages, key = { it.id }) { message ->
                                MessageBubble(message = message, isMe =
message.senderId == currentUserId)
                            }
                        }
                    }
                }
            }

            // --- Sync Status Indicator ---
            Text(
                text = uiState.syncStatus,
                style = MaterialTheme.typography.caption,
                modifier = Modifier
                    .fillMaxWidth()
```

```
                    .background(MaterialTheme.colors.surface)
                    .padding(horizontal = 16.dp, vertical = 4.dp),
                color = Color.Gray
            )

            // --- Message Input Bar ---
            Row(
                modifier = Modifier
                    .fillMaxWidth()
                    .background(MaterialTheme.colors.surface)
                    .padding(8.dp),
                verticalAlignment = Alignment.CenterVertically
            ) {
                OutlinedTextField(
                    value = uiState.currentInput,
                    onValueChange = viewModel::onInputChanged, // UI sends event
                    label = { Text("Type a message") },
                    modifier = Modifier.weight(1f),
                    singleLine = true,
                    enabled = !uiState.isSending // Disable input while sending
                )
                Spacer(modifier = Modifier.width(8.dp))
                Button(
                    onClick = viewModel::onSendClicked, // UI sends event
                    enabled = uiState.currentInput.isNotBlank() &&
!uiState.isSending, // Enable only if input exists and not sending
                    modifier = Modifier.height(56.dp) // Match TextField height
                ) {
                    if (uiState.isSending) {
                        CircularProgressIndicator(color =
MaterialTheme.colors.onPrimary, modifier = Modifier.size(24.dp))
                    } else {
                        Icon(Icons.Filled.Send, contentDescription = "Send
Message")
                    }
                }
            }
        }
    }
}

@Composable
fun MessageBubble(message: Message, isMe: Boolean) {
    val bubbleColor = if (isMe) MaterialTheme.colors.primary else
MaterialTheme.colors.secondary
    val textColor = if (isMe) MaterialTheme.colors.onPrimary else
MaterialTheme.colors.onSecondary
```

```
    val alignment = if (isMe) Alignment.End else Alignment.Start

    Column(
        modifier = Modifier.fillMaxWidth(),
        horizontalAlignment = alignment
    ) {
        Card(
            shape = RoundedCornerShape(8.dp),
            backgroundColor = bubbleColor,
            elevation = 2.dp
        ) {
            Column(modifier = Modifier.padding(8.dp)) {
                Text(
                    text = message.text,
                    color = textColor,
                    style = MaterialTheme.typography.body1
                )
                Spacer(modifier = Modifier.height(4.dp))
                val timeFormat = SimpleDateFormat("HH:mm", Locale.getDefault())
                Text(
                    text = "${message.senderId} -
${timeFormat.format(message.timestamp)}",
                    color = textColor.copy(alpha = 0.7f),
                    style = MaterialTheme.typography.caption
                )
                if (isMe) {
                    Text(
                        text = when (message.status) {
                            MessageStatus.SENT_OR_PENDING -> "Sending..."
                            MessageStatus.SENT_TO_SERVER -> "Sent"
                            MessageStatus.FAILED_TO_SEND -> "Failed"
                        },
                        color = textColor.copy(alpha = 0.7f),
                        style = MaterialTheme.typography.caption
                    )
                }
            }
        }
    }
}
```

Explanation:

- `ChatScreen` observes `uiState` and `sideEffects` from the `ViewModel`.
- It uses `LazyColumn` for efficient list display.
- `MessageBubble` is a helper `Composable` to display individual messages,

adapting appearance based on `isMe` and `message.status`.
- The UI shows loading, error, empty states, and the list of messages based on `uiState`.
- The input field and send button are wired to `viewModel::onInputChanged` and `viewModel::onSendClicked`.
- `LaunchedEffect(uiState.messages.size)` handles auto-scrolling to the bottom when new messages arrive.
- The `syncStatus` from `uiState` is displayed at the bottom.

Step 7.7: Main Activity (App Setup)

This is the entry point of your Android app.

File: MainActivity.kt

```
package com.yourcomapny.chatapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Scaffold
import androidx.compose.material.SnackbarHostState
import androidx.compose.material.rememberSnackbarHostState
import androidx.compose.runtime.remember
import com.yourcomapny.chatapp.presentation.chat.ChatScreen
import com.yourcomapny.chatapp.presentation.chat.ChatViewModelFactory
import androidx.lifecycle.viewmodel.compose.viewModel // For getting ViewModel in
Composable
import com.yourcomapny.chatapp.data.local.AppDatabase // For database
initialization

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Initialize Room Database (do this once in Application class in real app)
        // AppDatabase.getDatabase(applicationContext)

        setContent {
            MaterialTheme { // Your app's theme
                val snackbarHostState = remember { SnackbarHostState() }

                // Provide ViewModel via factory (passing applicationContext for
WorkManager)
```

```
                val chatViewModel:
com.yourappname.chatapp.presentation.chat.ChatViewModel = viewModel(
                    factory = ChatViewModelFactory(applicationContext)
                )

                ChatScreen(
                    viewModel = chatViewModel,
                    snackbarHostState = snackbarHostState,
                    currentUserId = "myUserId" // Pass current user ID
                )
            }
        }
    }
}
```

Explanation:

- `MainActivity` sets up the Compose content.
- It initializes the `ChatViewModel` using the `ChatViewModelFactory` (which provides the necessary dependencies like `ChatRepository`).
- It passes the `SnackbarHostState` down to `ChatScreen` for showing messages.
- This comprehensive setup provides a solid foundation for an Offline-First Chat App, demonstrating the integration of `Room`, `WorkManager`, `Flows`, `Coroutines`, and `Compose` within a clear architectural structure.