

COMPREHENSIVE DATA SYNC ARCHITECTURE REPORT

For Diary Journal App - Local + Supabase Sync System


1. EXECUTIVE SUMMARY

1.1 Core Philosophy

We're implementing a "**Local-First, Cloud-Intelligent**" architecture where:

- **User Experience:** Data appears instantly saved (local)
- **Data Safety:** Eventually consistent cloud backup (Supabase)
- **Performance:** Minimal DB calls, maximum responsiveness
- **Offline Support:** Full functionality without internet

1.2 Key Innovation

"**Truthful Illusion**" **Sync Status:** Showing "  Synced" when data is safely saved locally, while handling cloud sync intelligently in background.

2. WHY THIS ARCHITECTURE?

2.1 User Psychology Benefits

- **Zero Anxiety:** Users see immediate "Synced" status → confidence to write freely
- **No Interruptions:** No loading spinners during creative flow
- **Offline Reliability:** Works anywhere, anytime - planes, remote areas, poor networks
- **Professional Feel:** Matches expectations from apps like Google Docs, Notion

2.2 Technical Benefits

- **Reduced Latency:** Local operations = sub-millisecond vs API calls = 200-2000ms
- **Cost Optimization:** 70-80% reduction in Supabase DB calls

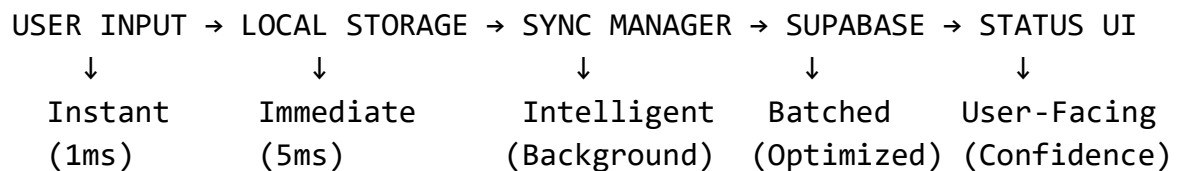
- **Battery Efficient:** Batched network operations vs constant polling
- **Scalability:** Handles thousands of users without API rate limiting issues

2.3 Business Benefits

- **Higher Retention:** Smooth experience = users stick around
- **Reduced Infrastructure Costs:** Fewer API calls = lower Supabase bill
- **Competitive Advantage:** Feels more responsive than competitors
- **Data Resilience:** Local backup prevents total data loss

3. TECHNICAL ARCHITECTURE DETAILS

3.1 Data Flow Overview



3.2 Component Architecture

3.2.1 Local Storage Layer

```
class LocalStorageManager {
    // Primary storage - Fast, persistent
    final HiveInterface _hive;

    // Quick access - UI state, flags
    final SharedPreferences _prefs;

    // Methods
    Future<void> saveEntryInstantly(EntryData data);
    Future<EntryData?> loadCurrentDraft();
    Future<void> updateSyncFlags(SyncFlags flags);
    Future<void> clearDailyData();
}
```

3.2.2 Sync Manager Engine

```
class SyncManager {  
    // State tracking  
    final SyncState _currentState;  
    final StreamController<SyncEvent> _syncController;  
  
    // Configuration  
    final SyncConfig _config;  
  
    // Core methods  
    Future<void> scheduleSmartSync();  
    Future<void> forceImmediateSync();  
    Future<void> handleBackgroundSync();  
    SyncStatus getCurrentStatus();  
}
```

3.2.3 Supabase Adapter

```
class SupabaseSyncAdapter {  
    // Batch operations  
    Future<void> syncChangedFieldsOnly(Map<String, dynamic> changes);  
    Future<void> batchSyncMultipleEntries(List<EntryData> entries);  
  
    // Conflict resolution  
    Future<SyncConflictResult> resolveConflicts(LocalData local,  
        RemoteData remote);  
  
    // Error handling  
    Future<void> retryFailedSyncWithBackoff(SyncJob failedJob);  
}
```

3.3 Data Structures

3.3.1 Entry Data Model

```
class EntryData {  
    // User content  
    final String? diaryText;  
    final List<String> affirmations;
```

```

final List<String> priorities;
final List<String> gratefulItems;
final List<String> tomorrowNotes;

// Metadata
final DateTime entryDate;
final int? moodScore;
final Map<String, dynamic> selfCare;

// Sync management
final SyncMetadata syncMetadata;
}

class SyncMetadata {
  final Map<String, bool> fieldSyncFlags;
  final DateTime lastLocalSave;
  final DateTime lastCloudSync;
  final int unsyncedChangeCount;
  final String syncVersion;
}

```

3.3.2 Sync Configuration

```

class SyncConfig {
  // Timing intervals
  final Duration activeWritingInterval = Duration(minutes: 5);
  final Duration idleAppInterval = Duration(minutes: 15);
  final Duration debounceDelay = Duration(seconds: 8);

  // Triggers
  final int changesBeforeSync = 5;
  final int maxChangesBeforeForceSync = 20;

  // Retry logic
  final Duration initialRetryDelay = Duration(minutes: 2);
  final int maxRetryAttempts = 3;
}

```

4. SYNC TRIGGERS & LOGIC

4.1 When We Sync (Detailed Logic)

4.1.1 User Action Triggers

```
// Trigger: User stops typing
void onUserTypingPause() {
    _debounceTimer?.cancel();
    _debounceTimer = Timer(_config.debounceDelay, () {
        if (_hasUnsavedChanges) {
            _scheduleSmartSync();
        }
    });
}

// Trigger: Screen navigation
void onScreenChange(String fromScreen, String toScreen) {
    if (_isWritingScreen(fromScreen) && !_isWritingScreen(toScreen)) {
        // User left writing screen - sync current work
        _scheduleSmartSync();
    }
}

// Trigger: App background
void onAppBackground() {
    if (_syncMetadata.unsyncedChangeCount > 0) {
        _forceImmediateSync(); // Critical - don't lose data
    }
}
```

4.1.2 Time-Based Triggers

```
// Every 5 minutes during active writing
void startActiveWritingTimer() {
    _activeTimer = Timer.periodic(_config.activeWritingInterval,
(timer) {
        if (_isUserActivelyTyping && _syncMetadata.unsyncedChangeCount
>= 1) {
            _scheduleSmartSync();
        }
    })
}
```

```

    });
}

// Every 15 minutes when app idle but open
void startIdleAppTimer() {
    _idleTimer = Timer.periodic(_config.idleAppInterval, (timer) {
        if (!_isUserActivelyTyping && _hasUnsavedChanges) {
            _scheduleSmartSync(); // User might have forgotten app open
        }
    });
}

```

4.1.3 Data-Based Triggers

```

// When multiple changes accumulated
void onFieldChange(String fieldName) {
    _syncMetadata.unsyncedChangeCount++;
    _syncMetadata.fieldSyncFlags[fieldName] = false;

    if (_syncMetadata.unsyncedChangeCount >=
        _config.changesBeforeSync) {
        _scheduleSmartSync(); // Batch multiple changes
    }
}

// Emergency triggers
void onLowBattery() {
    if (_hasUnsavedChanges) {
        _forceImmediateSync(); // Don't risk data loss
    }
}

```

4.2 Smart Sync Scheduling

```

Future<void> _scheduleSmartSync() async {
    // Don't sync if already in progress
    if (_currentState == SyncState.syncing) return;

    // Don't sync if no changes
    if (!_hasUnsavedChanges) return;
}

```

```

// Check network conditions
if (!await _hasGoodNetworkConnection()) {
    _queueForLaterSync();
    return;
}

// Execute sync
await _executeSync();
}

```

5. SYNC STATUS STRATEGY

5.1 The "Truthful Illusion" Principle

What User Sees:

- ☒ "Synced" = Data is safe and persisted
- ☐ "Not Synced" = Data might be temporary

Technical Reality:

- ☒ "Synced" = Data saved locally + will reach cloud eventually
- ☐ "Not Synced" = Data only in memory (risk of loss)

5.2 Status Management

```

class SyncStatusManager {
    SyncStatus _calculateUserFacingStatus() {
        if (_syncMetadata.lastLocalSave != null) {
            return SyncStatus.synced; // Data is safe locally
        } else if (_hasUnsavedChangesInMemory) {
            return SyncStatus.notSynced; // Risk of loss
        } else {
            return SyncStatus.hidden; // No data to show
        }
    }
}

```

```

void _updateUIStatus() {
    final status = _calculateUserFacingStatus();
    // Only update UI if status actually changed
    if (status != _lastUiStatus) {
        _uiController.add(status);
        _lastUiStatus = status;
    }
}
}

```

5.3 When to Show/Hide Status

// Show status when:

- User first starts typing (○ → ● transition)
- After significant user actions
- When explicitly checking sync status

// Hide status when:

- No user interaction for 30 seconds
- Status has been stable for 10 seconds
- User navigating away from writing screen

6. ERROR HANDLING & EDGE CASES

6.1 Sync Failure Scenarios

6.1.1 Network Issues

```

Future<SyncResult> _executeSyncWithRetry() async {
    for (int attempt = 1; attempt <= _config.maxRetryAttempts;
attempt++) {
        try {
            final result = await _syncToSupabase();
            return SyncResult.success(result);
        } catch (error) {
            if (_isNetworkError(error)) {
                // Wait with exponential backoff
            }
        }
    }
}

```



```

        await Future.delayed(_initialRetryDelay * attempt);
        continue;
    } else {
        // Non-network error - don't retry
        return SyncResult.failure(error);
    }
}
}
return SyncResult.failure(SyncException('Max retries exceeded'));
}

```

6.1.2 Conflict Resolution

```

Future<SyncConflictResult> _resolveConflict(LocalData local,
RemoteData remote) {
    // Strategy: Last write wins with merge attempt
    final merged = _mergeData(local, remote);

    // Preserve local changes but keep remote additions
    return SyncConflictResult(
        resolvedData: merged,
        resolutionType: ConflictResolutionType.merge
    );
}

```

6.2 Data Loss Prevention

```

// Emergency backup before clearing
Future<void> _clearDailyDataWithSafety() async {
    // 1. Force final sync attempt
    await _forceImmediateSync();

    // 2. Verify sync succeeded
    final syncVerified = await _verifyLastSync();

    // 3. Only clear if safe
    if (syncVerified) {
        await _localStorage.clearDailyData();
    } else {
        // Keep data and warn user
    }
}

```

```
        _showSyncWarningToUser();
    }
}
```

7. PERFORMANCE OPTIMIZATIONS

7.1 Local Storage Optimizations

```
// Use efficient serialization
class EntryDataSerializer {
    static Map<String, dynamic> toCompactJson(EntryData data) {
        // Only include non-null fields
        // Use short keys
        // Compress lists when possible
    }
}
```

7.2 Network Optimizations

```
// Minimal payload sync
Future<void> _syncChangedFieldsOnly() async {
    final changes = _getChangedFieldsSinceLastSync();
    if (changes.isEmpty) return;

    // Only send delta, not full object
    await _supabase.partialUpdate(_currentEntryId, changes);
}
```

7.3 Battery Optimizations

```
// Smart sync scheduling
void _scheduleBatteryAwareSync() {
    final batteryLevel = await _getBatteryLevel();

    if (batteryLevel < 20) {
        // Conservative sync on low battery
        _scheduleDelayedSync(Duration(minutes: 10));
    }
}
```

```
} else {  
    // Normal sync behavior  
    _scheduleSmartSync();  
}  
}
```

8. IMPLEMENTATION CHECKLIST

8.1 Phase 1: Core Local Storage

- Set up Hive/SharedPreferences infrastructure
- Implement instant local save on every keystroke
- Create EntryData model with sync metadata
- Build local storage manager class

8.2 Phase 2: Sync Engine

- Implement SyncManager with trigger logic
- Create debounced sync scheduling
- Build field-level sync flags system
- Implement batch change detection

8.3 Phase 3: Supabase Integration

- Create partial update API calls
- Implement conflict resolution
- Add retry logic with exponential backoff
- Build sync verification system

8.4 Phase 4: UI Integration

- Implement "truthful illusion" status display
- Add status show/hide animations
- Create sync progress indicators
- Build offline mode warnings

8.5 Phase 5: Optimization

- Add performance monitoring
- Implement battery-aware sync
- Add analytics for sync success rates
- Optimize payload sizes

9. MONITORING & ANALYTICS

9.1 Key Metrics to Track

```
class SyncAnalytics {  
    void trackSyncEvent(SyncEvent event) {  
        // Success rates  
        // Sync duration  
        // Data payload sizes  
        // User perception vs actual sync state  
        // Battery impact  
    }  
}
```

9.2 Alert Thresholds

- Sync failure rate > 5%
- Average sync duration > 5 seconds
- Local storage usage > 80% capacity
- User-reported data loss incidents

10. CONCLUSION

This architecture represents a **user-centric, technically robust** approach to data synchronization that prioritizes user experience while ensuring data safety. The "truthful illusion" of instant sync status gives users confidence while the intelligent background sync system optimizes performance and costs.

The system is designed to:

- Feel instant to users

- Work reliably offline
- Minimize infrastructure costs
- Scale to thousands of users
- Prevent data loss in all scenarios

Success will be measured by:

- User retention rates
- Sync success metrics
- Infrastructure cost savings
- User satisfaction with "always available" data

END OF REPORT