GatherSync Offline-First Sync Architecture

Version: 2.0

Date: December 21, 2024

Status: Design Proposal

Core Principles

1. Local Storage is Source of Truth

Rule: All data operations happen on local storage first, always.

- Create → Save to AsyncStorage immediately
- Read → Always read from AsyncStorage
- Update → Update AsyncStorage first
- Delete → Delete from AsyncStorage first

Why: Guarantees instant response, works offline, never loses data.

2. Cloud is Sync Layer Only

Rule: Cloud storage mirrors local data, never replaces it.

- Cloud sync happens in background
- Cloud failures don't block user operations
- Cloud is backup + cross-device sync
- Local data survives login/logout

Why: Reliability, offline support, no data loss from sync failures.

3. Never Hide Local Data

Rule: Login state never makes local data invisible.

- Logged out: Show local data
- Logged in: Show merged local + cloud data
- Switching states: Preserve all data
- No data disappears when logging in/out

Why: Prevents the data loss we experienced.

4. Explicit Sync Operations

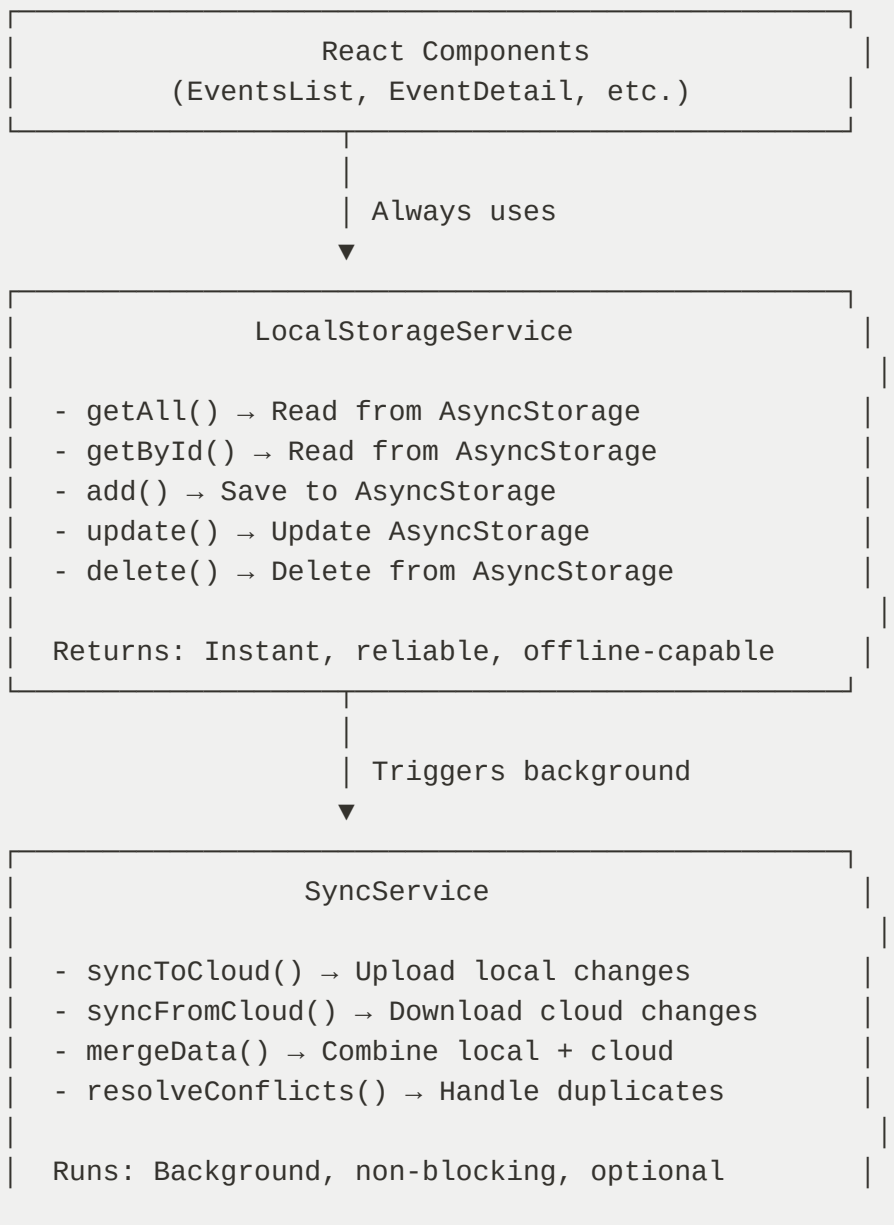
Rule: User controls when sync happens.

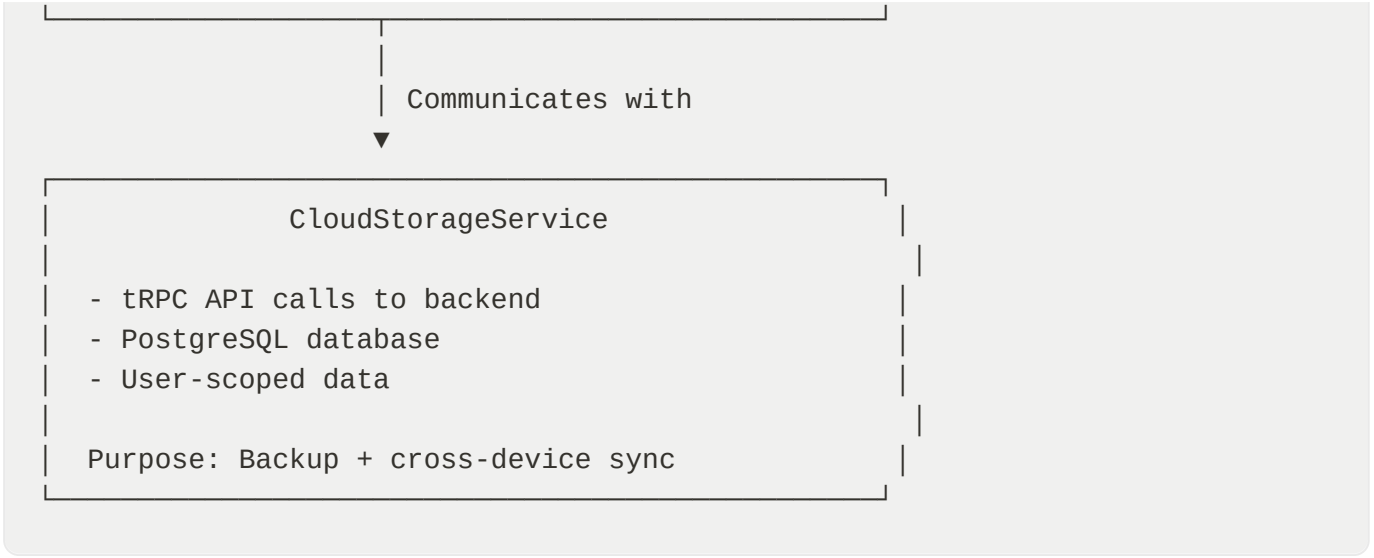
- Manual "Sync Now" button
- Clear sync status indicators
- Show what will sync before syncing
- Confirm before overwriting conflicts

Why: User understands what's happening, no surprises.

Architecture Overview

Plain Text





Data Flow

Creating an Event

User Action: Tap "Create Event", fill form, tap "Save"

Flow:

1. **LocalStorageService.add(event)**
   - Generate unique ID
   - Add timestamp
   - Save to AsyncStorage
   - Return immediately ✓
2. **UI updates instantly** with new event
3. **SyncService.syncToCloud()** (background)
   - Check if authenticated
   - If yes: Upload event to cloud
   - If no: Skip (will sync on next login)
   - If fails: Retry later, local data safe

Result: User sees event immediately, cloud sync happens later.

Editing an Event

User Action: Tap event, tap edit, change fields, tap "Save"

Flow:

1. **LocalStorageService.update(id, changes)**
   - Find event in AsyncStorage
   - Merge changes
   - Save updated event
   - Return immediately ✓
2. **UI updates instantly** with changes
3. **SyncService.syncToCloud()** (background)
   - Upload updated event to cloud
   - Mark as synced
   - If fails: Mark as pending sync

Result: Changes visible immediately, sync happens in background.

Logging In

User Action: Tap "Log In", complete OAuth, return to app

Flow:

1. **Save authentication token**
2. **SyncService.syncFromCloud()**
   - Download all cloud events for this user
   - Compare with local events
   - Merge strategy:
     - Cloud event not in local → Add to local
     - Local event not in cloud → Upload to cloud
     - Same event in both → Keep newer (by updatedAt)
     - Conflicts → Ask user
3. **LocalStorageService.getAll()**
   - Read merged data from AsyncStorage
   - Display to user
4. **Enable background sync**
   - Future changes auto-sync to cloud

Result: Local + cloud data merged, nothing lost.

Logging Out

User Action: Tap "Log Out"

Flow:

1. **SyncService.syncToCloud()** (final sync)
   - Upload any pending local changes
   - Wait for completion
2. **Clear authentication token**
3. **Keep all local data** in AsyncStorage
4. **Disable background sync**
5. **LocalStorageService.getAll()**
   - Still shows all events (from local)

Result: Data stays on device, available offline.

Storage Schema

AsyncStorage Keys

TypeScript

```
// Events
@gathersync_events: Event[]

// Sync metadata
@gathersync_sync_status: {
  lastSyncTime: string;
  pendingChanges: string[]; // Event IDs
  conflicts: Conflict[];
}

// User session
@gathersync_auth: {
  token: string;
  user: User;
  expiresAt: string;
}
```

Event Object

TypeScript

```
interface Event {
  // Core fields
  id: string;              // UUID
  name: string;
  eventType: 'flexible' | 'fixed';

  // Timing
  month: number;           // 1-12
  year: number;
  fixedDate?: string;      // ISO date for fixed events
  fixedTime?: string;      // HH:MM for fixed events

  // Participants
  participants: Participant[];

  // Meeting details
  teamLeader?: string;
  teamLeaderPhone?: string;
  meetingType?: 'in-person' | 'virtual';
  venueName?: string;
  venueContact?: string;
  venuePhone?: string;
  meetingLink?: string;
  rsvpDeadline?: string;
  meetingNotes?: string;

  // Metadata
  createdAt: string;       // ISO timestamp
  updatedAt: string;       // ISO timestamp
  syncedAt?: string;       // Last cloud sync time

  // Sync tracking
  pendingSync: boolean;    // Has local changes not synced
  cloudId?: string;        // ID in cloud database (may differ from local)
}
```

Implementation Plan

Phase 1: Local Storage Service (Priority 1)

File: `lib/local-storage.ts`

Functions:

TypeScript

```
class LocalStorageService {
  private static EVENTS_KEY = '@gathersync_events';

  // Read operations
  async getAll(): Promise<Event[]>
  async getById(id: string): Promise<Event | null>
  async search(query: string): Promise<Event[]>

  // Write operations
  async add(event: Omit<Event, 'id' | 'createdAt' | 'updatedAt'>):
Promise<Event>
  async update(id: string, changes: Partial<Event>): Promise<Event>
  async delete(id: string): Promise<void>

  // Batch operations
  async addMany(events: Event[]): Promise<void>
  async updateMany(updates: Array<{id: string, changes: Partial<Event>}>):
Promise<void>

  // Utility
  async clear(): Promise<void>
  async export(): Promise<string> // JSON backup
  async import(data: string): Promise<void>
}
```

Testing:

- Unit tests for each method
- Test with empty storage
- Test with existing data
- Test error handling
- Test data integrity

Phase 2: Sync Service (Priority 2)

File: `lib/sync-service.ts`

Functions:

TypeScript

```
class SyncService {
  // Main sync operations
  async syncToCloud(): Promise<SyncResult>
  async syncFromCloud(): Promise<SyncResult>
```

```
  async fullSync(): Promise<SyncResult>

  // Conflict resolution
  async detectConflicts(): Promise<Conflict[]>
  async resolveConflict(conflict: Conflict, resolution: 'local' | 'cloud' |
'merge'): Promise<void>

  // Status
  async getSyncStatus(): Promise<SyncStatus>
  async getPendingChanges(): Promise<Event[]>

  // Background sync
  startAutoSync(intervalMs: number): void
  stopAutoSync(): void
}

interface SyncResult {
  success: boolean;
  uploaded: number;
  downloaded: number;
  conflicts: number;
  errors: string[];
}

interface SyncStatus {
  lastSyncTime: Date | null;
  pendingChanges: number;
  isOnline: boolean;
  isAuthenticated: boolean;
}
```

Testing:

- Test upload to cloud
- Test download from cloud
- Test merge logic
- Test conflict detection
- Test offline behavior
- Test authentication failures

Phase 3: UI Integration (Priority 3)

Changes to Components:

1. Replace eventsStorage with localStorageService

TypeScript

```
// Old (broken)
import { eventsStorage } from '@/lib/hybrid-storage';

// New (reliable)
import { localStorageService } from '@/lib/local-storage';

// Usage stays the same
const events = await localStorageService.getAll();
```

2. Add Sync UI

New components:

- `SyncStatusBanner` - Shows sync status, last sync time
- `SyncButton` - Manual sync trigger
- `ConflictResolver` - UI for resolving conflicts

3. Update Event Screens

- Events list: Add sync status indicator
- Event detail: Show if event has pending changes
- Create/Edit: Mark as pending sync after save

Phase 4: Migration (Priority 4)

Goal: Move existing data (if any) to new system

File: `lib/migration.ts`

TypeScript

```
async function migrateToNewStorage() {
  // 1. Check if old hybrid storage has data
  const oldData = await checkOldStorage();

  // 2. If found, migrate to new local storage
  if (oldData.length > 0) {
    await localStorageService.addMany(oldData);
  }

  // 3. Mark migration complete
  await AsyncStorage.setItem('@gathersync_migrated', 'true');
}
```

Run: On app launch, one time only

Sync Strategies

Initial Sync (First Login)

Scenario: User has local events, logs in for first time

Strategy:

1. Upload all local events to cloud
2. Download all cloud events (should be empty)
3. Mark all as synced
4. Enable auto-sync

Result: Local data preserved, now backed up to cloud

Subsequent Syncs

Scenario: User has been using app, syncs again

Strategy:

1. **Upload phase:**
   - Find events with `pendingSync: true`
   - Upload to cloud
   - Update `syncedAt` timestamp
   - Mark `pendingSync: false`
2. **Download phase:**
   - Get all cloud events
   - For each cloud event:
     - If not in local: Add to local
     - If in local but cloud newer: Update local
     - If in local but local newer: Skip (already uploaded)
3. **Cleanup:**
   - Update last sync time
   - Clear pending changes list

Conflict Resolution

Conflict Types:

1. Same event edited on multiple devices

Detection:

- Event exists in both local and cloud
- Both have `updatedAt` after last sync
- Changes to same fields

Resolution options:

- Keep local (default)
- Keep cloud
- Merge (combine non-conflicting fields)
- Show user both versions, let them choose

2. Event deleted locally but edited in cloud

Detection:

- Event in cloud but not in local
- Cloud `updatedAt` after last sync
- No deletion record locally

Resolution:

- Assume local deletion intentional
- Delete from cloud
- OR: Ask user if they want to restore

3. Event deleted in cloud but edited locally

Detection:

- Event in local but not in cloud
- Local `updatedAt` after last sync
- Cloud returns 404

Resolution:

- Re-upload to cloud (local version wins)

Error Handling

Network Errors

Scenario: Sync fails due to no internet

Handling:

- Show "Offline" indicator
- Keep local changes
- Mark as pending sync
- Retry automatically when online

User sees: "Working offline. Changes will sync when online."

Authentication Errors

Scenario: Token expired, user logged out remotely

Handling:

- Show "Session expired" message
- Prompt to log in again
- Keep local data safe
- Don't attempt sync until re-authenticated

User sees: "Please log in again to sync"

Cloud Storage Errors

Scenario: Cloud database error, API failure

Handling:

- Log error details
- Show user-friendly message
- Keep local data safe
- Mark as pending sync
- Retry with exponential backoff

User sees: "Sync failed. Will retry automatically."

Data Validation Errors

Scenario: Local event missing required fields for cloud

Handling:

- Add default values for missing fields
- Log which fields were added
- Upload with defaults
- Don't block sync for other events

User sees: Sync succeeds, no error shown

Testing Strategy

Unit Tests

Local Storage Service:

- ✓ Create event
- ✓ Read event
- ✓ Update event
- ✓ Delete event
- ✓ Handle missing data
- ✓ Handle corrupted data
- ✓ Export/import

Sync Service:

- ✓ Upload to cloud
- ✓ Download from cloud
- ✓ Merge local + cloud
- ✓ Detect conflicts
- ✓ Resolve conflicts
- ✓ Handle offline
- ✓ Handle auth errors

Integration Tests

Full Sync Flow:

- ✓ Create event offline
- ✓ Log in
- ✓ Sync to cloud

- ✓ Verify in cloud database
- ✓ Log out
- ✓ Verify local data still present

Cross-Device Sync:

- ✓ Create event on device A
- ✓ Sync to cloud
- ✓ Log in on device B
- ✓ Sync from cloud
- ✓ Verify event appears on device B

Conflict Resolution:

- ✓ Edit same event on two devices offline
- ✓ Sync both
- ✓ Detect conflict
- ✓ Resolve with user choice
- ✓ Verify resolution propagates

Manual Testing Checklist

Before Deployment:

- [ ] Create event while logged out → Appears in list
- [ ] Edit event while logged out → Changes save
- [ ] Delete event while logged out → Disappears
- [ ] Log in → Events still visible
- [ ] Sync → Events upload to cloud
- [ ] Check admin dashboard → Events appear
- [ ] Log out → Events still visible locally
- [ ] Create event while logged in → Saves locally and syncs
- [ ] Edit event while logged in → Saves locally and syncs
- [ ] Turn off WiFi → App still works
- [ ] Make changes offline → Marked as pending
- [ ] Turn on WiFi → Auto-sync uploads changes

- [ ] Force close app → Data persists on reopen
- [ ] Uninstall and reinstall → Data lost (expected)

Migration Path

From Current Broken State

Step 1: Backup any remaining data

- Export from admin dashboard if possible
- Save to JSON file

Step 2: Deploy new storage system

- Replace hybrid-storage.ts with local-storage.ts
- Add sync-service.ts
- Update all imports in components

Step 3: Test thoroughly

- Run all unit tests
- Run integration tests
- Manual testing checklist

Step 4: Deploy to user

- Create checkpoint
- User tests on iPhone
- Verify create/edit/delete work
- Verify data persists

Step 5: Enable cloud sync

- User logs in
- Triggers initial sync
- Verify data in cloud
- Enable auto-sync

Success Criteria

Must Work

- ✓ Create event → Saves immediately, visible in list
- ✓ Edit event → Changes persist after app restart
- ✓ Delete event → Removed from list permanently
- ✓ Works offline → All features functional without internet
- ✓ Login → Doesn't lose local data
- ✓ Logout → Doesn't lose local data
- ✓ Sync → Uploads local events to cloud
- ✓ Sync → Downloads cloud events to local
- ✓ Multiple devices → Events sync between devices

Performance

- ✓ Create/edit/delete → Instant (< 100ms)
- ✓ Load events list → Fast (< 500ms)
- ✓ Sync → Background, non-blocking
- ✓ Large datasets → Handles 100+ events smoothly

Reliability

- ✓ No data loss under any circumstances
- ✓ Sync failures don't break app
- ✓ Network errors handled gracefully
- ✓ Works after app updates
- ✓ Works after device restart

Timeline

Day 1: Foundation

- Implement LocalStorageService
- Write unit tests
- Test in isolation

Day 2: Sync Layer

- Implement SyncService
- Write integration tests
- Test sync flows

Day 3: UI Integration

- Update all components
- Add sync UI
- Manual testing

Day 4: Polish & Deploy

- Fix any bugs found
- Final testing checklist
- Deploy to user
- User acceptance testing

Total: 4 days of focused development

Conclusion

This architecture prioritizes reliability and user trust over clever features. By keeping local storage as the source of truth and treating cloud as a sync layer, we eliminate the data loss scenarios that plagued the previous implementation.

The offline-first approach means users can always access and modify their data, regardless of network conditions or authentication state. Sync happens in the background, transparently, without blocking user operations.

This is the proven pattern used by professional apps like Notion, Todoist, and Evernote. It works.

Next Step: Review this design, get approval, then implement Phase 1 (Local Storage Service).