Raymond Bautista

Southern New Hampshire University (SNHU)

CS-499: Computer Science Capstone

April 5, 2026

Milestone Four: Databases Enhancement

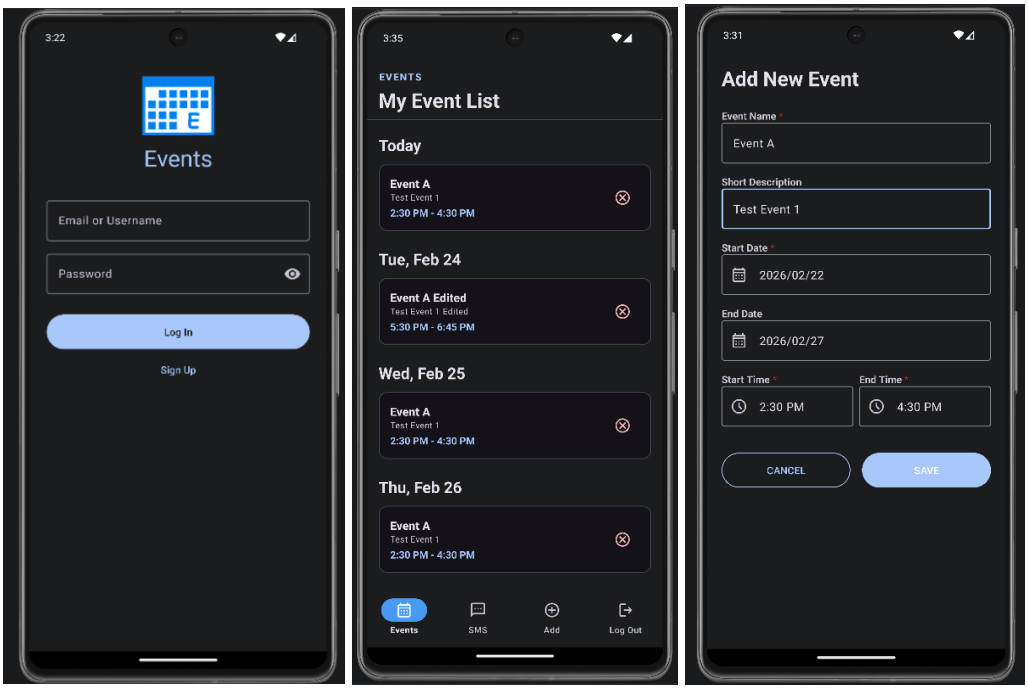
Artifact Overview

For this category, I selected the Events Android application developed in CS-360: Mobile Architecture and Programming (2026-C1). This project is a user-centered mobile application built using Android Studio, the Java programming language, and a local SQLite database. It emphasizes modern UI/UX best practices, lifecycle-aware architecture, secure authentication, and efficient performance, with the goal of reducing user cognitive load while delivering a smooth and distraction-free experience. The application functions as a personal scheduling hub, presenting events in an agenda-style list grouped by date and sorted in ascending chronological order, enabling users to easily track and manage multi-day commitments.

I selected this artifact because its initial implementation relied on a local native (Room) database on the Android device using SQLite, following a relational database model. The schema consisted of two tables: users and events, where events were retrieved in ascending chronological order and filtered by the user_id foreign key. While this design is effective for single-device usage, it introduces a significant limitation: if the same user attempts to log in from a different device, the application cannot recognize their credentials or associated event data. Consequently,

even if the user creates a new account on another device, their events will not be available, as the data remains stored locally, resulting in fragmented and inconsistent datasets.

To address this limitation, I conducted research on various cloud-based database solutions and architectural approaches to transition the application from a local relational model to a cloud-hybrid architecture. In this design, a cloud-based document-oriented (NoSQL) database serves as the primary source of truth, storing and managing data for all users across devices. This central repository synchronizes with the local Room database, which functions as a caching layer to provide fast, offline-capable access to the currently authenticated user's data. This hybrid approach improves data consistency, scalability, and availability while maintaining high performance on the client side.



Proposed Enhancements

For this category, I have implemented the following improvements:

- Integration of a cloud-based, document-oriented database (Firestore) as the primary source of truth for all application data.
- Design of a data architecture that supports consistent user sessions across multiple devices.
- Implementation of a hybrid cloud architecture that enables offline functionality for authenticated users, along with automatic synchronization with cloud services.
- Application of security measures for real-time data access, ensuring that each user can only access their own data while maintaining the integrity and confidentiality of all user information stored in the cloud.

Enhancements Implementation Process and Learning Reflections

- Cloud Services Benchmark

For this cloud migration, I evaluated many Backend-as-a-Service (BaaS) alternatives that provide database integration alongside analytics, deployment tools, authentication, and other managed services. My research focused on solutions compatible with mobile development, particularly Android, including Google Firebase Firestore, Amazon AWS Amplify (DynamoDB), and MongoDB Atlas.

Amazon AWS Amplify offers enterprise-level scalability as part of the broader AWS ecosystem. It supports GraphQL via AppSync for real-time synchronization with NoSQL databases (key-value and document models), enabling efficient handling of large-scale data. Additionally, it provides a one-year free tier with up to 25 GB of storage, which is sufficient for initial testing. However, this option presents a steep learning curve, requiring proficiency with the Amplify CLI, AWS SDK, and complex IAM (Identity and Access Management)

configurations. These requirements make it less suitable for a solo project with limited development time (Amplify Hosting, 2026).

Another candidate was MongoDB Atlas, which I have previously used in full-stack applications with Java, Python, and JavaScript. It offers a permanent free tier of 512 MB on a shared cluster, with pricing based on usage (read/write operations) (MongoDB, n.d.). MongoDB provides a flexible, schema-less structure well-suited for complex JSON-based data models. However, real-time synchronization requires implementing Change Streams, which subscribe to data modifications at the collection or database level. While effective in web and desktop environments, this approach introduces additional complexity when integrating with mobile applications using Java, particularly in managing persistent listeners and efficient resource usage (Team, 2026).

The most suitable option for this project was Firebase Firestore, which stands out for its seamless integration with Android Studio and its design focus on mobile-first applications, while remaining cross-platform. This compatibility aligns well with the existing codebase, allowing integration with minimal refactoring and enabling a primary focus on refining the data layer. Firestore offers a generous free tier of 1 GB storage, 50,000 reads per day, and 20,000 writes per day, with usage-based pricing beyond these limits (Firebase, 2019). In addition to its flexible, document-oriented NoSQL structure, Firestore provides two key capabilities essential for this enhancement. First, it includes offline persistence by default, allowing the application to continue functioning during network interruptions by caching data locally and synchronizing queued operations once connectivity is restored (Enabling Offline Capabilities on Android | Firebase Realtime Database, n.d.). Second, it supports real-time synchronization through native listeners,

such as the onSnapshot() method, which delivers immediate data snapshots and automatically updates them when changes occur (Google, n.d.).

Based on these advantages, I selected Firebase Firestore due to its seamless mobile integration with Android Studio, ease of implementation without extensive refactoring, suitable free-tier capabilities for initial development, and built-in support for offline persistence and real-time synchronization.

- Synchronized Hybrid Cloud Architecture

For this enhancement, I implemented an offline-first architecture rather than relying exclusively on cloud-based operations. This approach enables the application to perform most of its critical functionality without an active internet connection (Build an Offline-First App, n.d.), while still leveraging continuous synchronization with the cloud to ensure data consistency across user devices. Specifically, I adopted a Cloud-Authoritative Hybrid Sync Pattern, where Firebase Firestore acts as the master source of truth, and the local Room database functions as a high-speed caching layer. The repository accesses Room as the device's canonical data source, eliminating network latency and significantly improving performance. As a result, local queries execute in tens of milliseconds compared to potentially hundreds of milliseconds for cloud-based operations, depending on network conditions.

Within this architecture, the UI layer (activities and intents) remains responsive by interacting exclusively with the local Room database through LiveData, without directly communicating with the cloud. The repository layer encapsulates the business logic, determining when to fetch data from Firestore and when to persist it locally (Build an Offline-First App, n.d.). The synchronization strategy follows a conditional workflow: when network connectivity is

available, updates are first applied to Firestore and then propagated to Room; otherwise, updates are written locally and queued for later synchronization with the cloud once connectivity is restored (Enabling Offline Capabilities on Android | Firebase Realtime Database, n.d.).

From a data management perspective, Firestore stores all user and event data as the centralized source of truth. To ensure security and data integrity, access is restricted so that each user can only retrieve their own data. On the client side, I implemented a user-scoped Room database strategy, where local tables are cleared upon logout and repopulated exclusively with data corresponding to the authenticated user upon login. This prevents data leakage between users on the same device.

Additionally, I leveraged SharedPreferences to persist the authenticated USER_ID, allowing the application to maintain session continuity and bypass the login process when appropriate. Upon launch, if a valid user session exists, the application navigates directly to the main screen and loads data from the local cache. The current limitation of this design is that, because the local database is cleared on logout, initial data retrieval after login requires network connectivity. Therefore, user authentication (login and registration) is dependent on an active internet connection to synchronize with the cloud.

- Code Implementation and Challenges

Initial Dependencies Conflicts Challenges

The first step in the cloud integration process involved modifying the Gradle project and module files to include the required Firestore dependencies, as well as adding the google-services.json file to the app directory. This configuration enables secure communication between the application and Firebase services by providing the necessary credentials and connection

settings. During the initial Gradle synchronization, I encountered the following compilation error:

> error: cannot access ListenableFuture
>
> class file for com.google.common.util.concurrent.ListenableFuture not found

This issue occurred because the SMS WorkManager component depended on a standalone version of ListenableFuture, while Firebase introduced the full Guava library through its Bill of Materials (BoM), which includes the same class. This conflict resulted in what is commonly referred to as *dependency hell*, a condition in which incompatible or overlapping dependencies cause build failures, runtime issues, or potential vulnerabilities (Sonar, 2019). The Firebase BoM is essential for managing compatible versions of Firebase libraries, but it can introduce transitive dependencies that conflict with existing project configurations (Firebase Android Release Notes, n.d.). To resolve this issue, I explicitly added the Guava library dependency: implementation("com.google.guava:guava:32.1.3-android"). This ensured that the required ListenableFuture class was properly recognized by the compiler. Additionally, I configured dependency exclusions to remove conflicting standalone versions of ListenableFuture that remained in the project, preventing duplicate dependency errors. This solution restored building stability and ensured compatibility between WorkManager and Firebase components.

```
57      // Explicitly add the full Guava library that Firebase uses.
58      // This "forces" the compiler to recognize the class for the Worker.
59      implementation("com.google.guava:guava:32.1.3-android")
60  }
61
62  // This tells Gradle to ignore the standalone 'listenablefuture' that causes the "Duplicate" error
63  configurations {
64      all*.exclude group: 'com.google.guava', module: 'listenablefuture'
65  }
```

Code Implementation

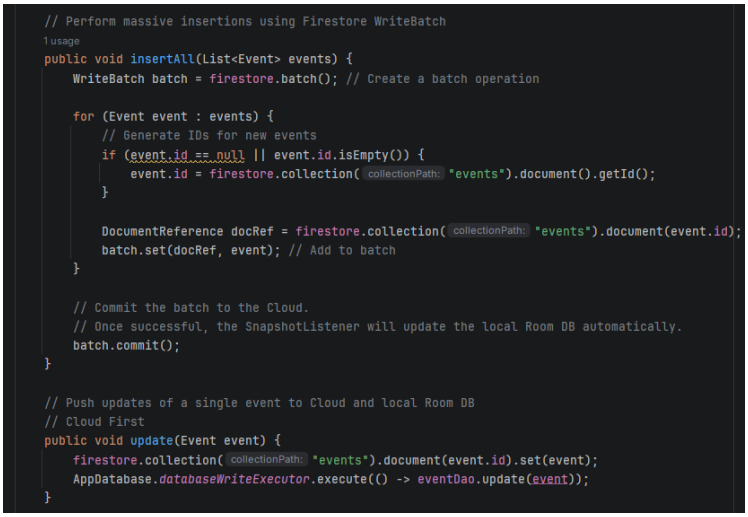
The changes to the data layer were minimal, as I adopted a hybrid cloud architecture in which the local Room database acts as the canonical data source on the device, while the cloud database serves as the remote master source of truth. In this design, the existing SQL-based entities, schemas, and queries were preserved, while NoSQL operations for cloud interaction were introduced within the repository layer, following the previously defined cloud-first synchronization strategy. The primary modification across the data layer involved refactoring identifier data types. Previously, the system relied on auto-incremented integer IDs; however, Firestore uses unique string-based document identifiers. To accommodate this, I updated the primary keys in both the User and Event model classes from integers to non-null string IDs. Additionally, I introduced no-argument (empty) constructors in these model classes to support proper object mapping and serialization required by Firestore.

```java
// Set the User table id as a foreign key for this model
// Apply cascading to delete the associated events if an user is deleted
@Entity(tableName = "events",
        foreignKeys = @ForeignKey(
                entity = User.class,
                parentColumns = "id",
                childColumns = "userId",
                onDelete = ForeignKey.CASCADE
        ),
        indices = {@Index("userId")})
public class Event {

    // Event table fields

    // Change from auto-incremental int ID to String Firestore ID
    @PrimaryKey
    @NonNull
    public String id;

    public String name;        // Obligatory
    15 usages
    public String description; // Optional
    public String date;        // YYYY/MM/DD - Obligatory
    16 usages
    public String startTime;   // HH:MM - Obligatory
    14 usages
    public String endTime;     // HH:MM - Obligatory

    // Changed to String ID to match Firestore UID
    @NonNull
    public String userId;         // Foreign Key
```

Within the Data Access Objects (DAOs), particularly in the EventDao, I configured the onConflict = OnConflictStrategy.REPLACE policy to ensure that cloud synchronization processes can safely overwrite the local cache when necessary. This approach supports consistency between the local and remote data sources during synchronization. At the repository layer, which acts as the bridge between the data layer and the ViewModels, I implemented a hybrid cloud synchronization method, startRealtimeSync. This method initializes an addSnapshotListener to observe changes in Firestore for events associated with the currently authenticated user ID. The listener operates asynchronously in the background, automatically propagating updates from the cloud to the local Room database, ensuring real-time consistency. Additionally, I refactored the CRUD operations to synchronize both the cloud master database and the local Room cache. For bulk operations, such as inserting recurring events, I optimized performance by leveraging Firestore's WriteBatch feature. This allows multiple write operations to be grouped into a single atomic request, ensuring that all operations either succeed or fail together, while reducing network overhead and improving efficiency.

```java
// Inserts a single event
// ADDED: onConflict = OnConflictStrategy.REPLACE so Cloud syncs can safely overwrite local cache
no usages  1 implementation
@Insert(onConflict = OnConflictStrategy.REPLACE)
void insert(Event event);
```

```java
// Hybrid-Cloud Sync: listens to the Firestore and updates local Room automatically
1 usage
public void startRealtimeSync(String userId) {
    firestore.collection( collectionPath: "events") CollectionReference
            .whereEqualTo( field: "userId", userId) Query
            .addSnapshotListener(( QuerySnapshot value, FirebaseFirestoreException error) -> {
                if (error != null || value == null) return;

                List<Event> cloudEvents = value.toObjects(Event.class);
                AppDatabase.databaseWriteExecutor.execute(() -> {
                    eventDao.deleteAllForUser(userId); // Clear old local cache
                    eventDao.insertAll(cloudEvents);   // Save fresh cloud data
                });
            });
}
```



```
// Perform massive insertions using Firestore WriteBatch
1 usage
public void insertAll(List<Event> events) {
    WriteBatch batch = firestore.batch(); // Create a batch operation

    for (Event event : events) {
        // Generate IDs for new events
        if (event.id == null || event.id.isEmpty()) {
            event.id = firestore.collection( collectionPath: "events").document().getId();
        }

        DocumentReference docRef = firestore.collection( collectionPath: "events").document(event.id);
        batch.set(docRef, event); // Add to batch
    }

    // Commit the batch to the Cloud.
    // Once successful, the SnapshotListener will update the local Room DB automatically.
    batch.commit();
}

// Push updates of a single event to Cloud and local Room DB
// Cloud First
public void update(Event event) {
    firestore.collection( collectionPath: "events").document(event.id).set(event);
    AppDatabase.databaseWriteExecutor.execute(() -> eventDao.update(event));
}
```

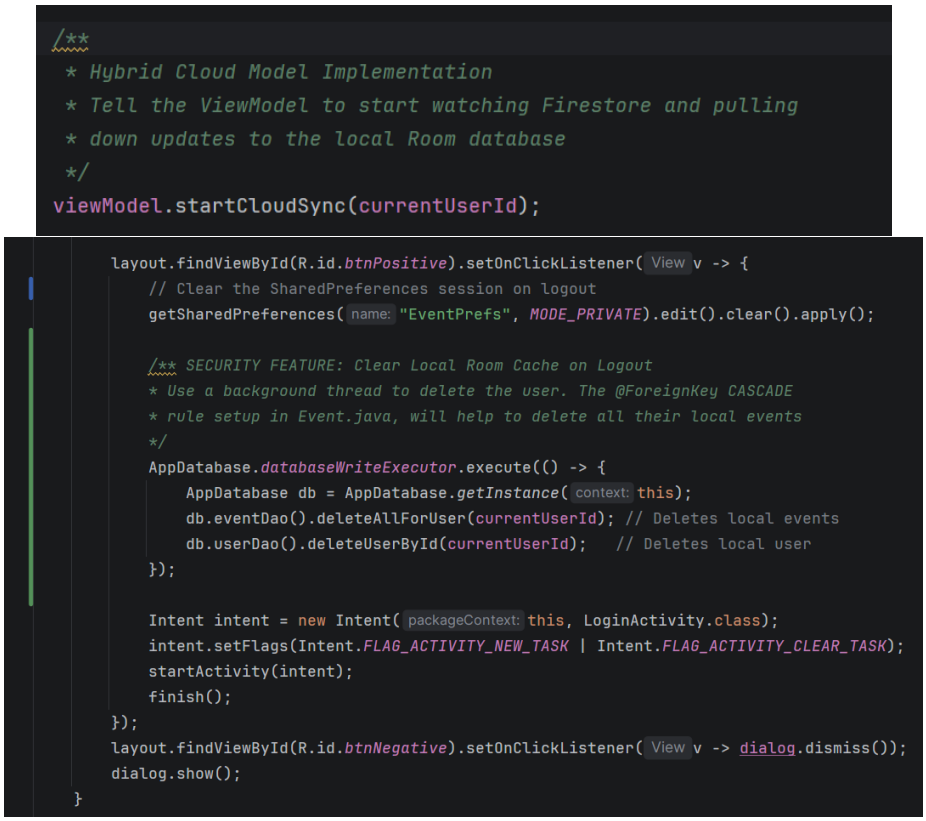
Within the User data repository, I updated the registration method to generate a unique string-based identifier using Firestore for each new user. The user credentials, including the hashed password, are then stored both in the local Room database and in the cloud master database to maintain consistency across devices. For the authentication process, I refactored the logic to securely validate user credentials against the cloud database in real time. This is achieved using addOnCompleteListener, which allows the application to authenticate users individually based on their provided credentials without retrieving the entire users collection from the NoSQL database. As a result, all authentication-related operations now rely on the cloudUser object obtained from the document query and are executed within the listener's callback, ensuring efficient, secure, and scalable authentication.

```java
/* Hashes user's password using BCrypt algorithm
 * applying salting for safe credential handling.
 * Then execute the service to register the user.
 */
1 usage
public void register(User user, OnRegisterListener listener) {
    AppDatabase.databaseWriteExecutor.execute(() -> {

        // Generate a unique String ID from Firestore for the new user
        String newUserId = firestore.collection( collectionPath: "users").document().getId();
        user.id = newUserId;

        // Hash the password with a generated salt before saving
        String hashedPassword = BCrypt.hashpw(user.password, BCrypt.gensalt());
        user.password = hashedPassword; // Overwrite plain text with hash

        // Saves locally for the current device
        userDao.insertUser(user);

        // Saves to Cloud Master Source of Truth
        firestore.collection( collectionPath: "users").document(user.id).set(user);

        if (listener != null) {
            listener.onFinished();
        }
    });
}
```

```java
public class UserRepository {
    * - Monitor and log failed authentication attempts for security auditing.
    * - Cloud First Approach: Safely authenticates on any device without downloading all users.
    */
    1 usage
    public void authenticate(String identifier, String password, OnAuthListener listener) {
        // Query the Cloud first to find the specific user by email or username
        firestore.collection( collectionPath: "users") CollectionReference
                .where(Filter.or(
                        Filter.equalTo( field: "email", identifier),
                        Filter.equalTo( field: "username", identifier)
                )) Query
                .limit(1) // Ensure we only get 1 result
                .get() Task<QuerySnapshot>
                .addOnCompleteListener( Task<QuerySnapshot> task -> {
                    if (task.isSuccessful() && !task.getResult().isEmpty()) {
                        // Extract ONLY this user's data from the cloud
                        User cloudUser = task.getResult().getDocuments().get(0).toObject(User.class);

                        // Perform security checks safely on a background thread
                        AppDatabase.databaseWriteExecutor.execute(() -> {
                            long now = System.currentTimeMillis();

                            // if the penalty time has expired, reset the brute-force counters
                            if (cloudUser.lockoutTimestamp > 0 && now > cloudUser.lockoutTimestamp) {
                                cloudUser.failedAttempts = 0;
                                cloudUser.lockoutTimestamp = 0;
                            }

                            // Check Brute-Force Attacks and implements a cooldown
                            if (cloudUser.lockoutTimestamp > now) {
                                long remaining = (cloudUser.lockoutTimestamp - now) / 60000;
                                listener.onFinished( user: null, statusMessage: "Account locked. Try again in " + (remaining + 1) + " mins.");
                                return;
                            }
```

The ViewModels and Activities required only minor modifications, as the primary focus of this database enhancement was on the data layer (models, DAOs, and repositories). The changes in these layers were mainly limited to adapting method signatures and updating data types to accommodate the transition from integer-based IDs to string-based identifiers. The most significant update was implemented in the MainActivity, where I invoked the startCloudSync method through the ViewModel to initiate real-time synchronization with Firestore. This enables

the application to continuously listen for remote updates and propagate them to the local Room database. Additionally, I incorporated a database write executor to handle background operations, specifically to trigger DAO methods that clear all user and event data from the local Room database upon logout. This ensures proper data isolation and prevents residual data from persisting across user sessions.



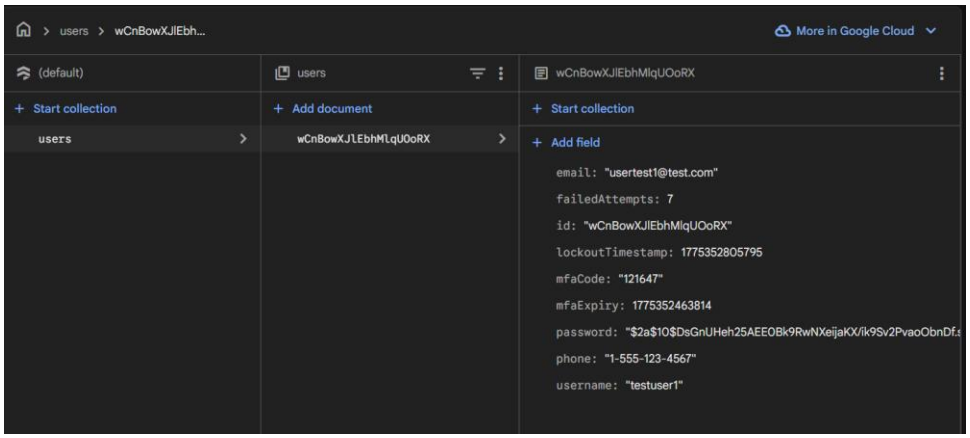
```
/**
 * Hybrid Cloud Model Implementation
 * Tell the ViewModel to start watching Firestore and pulling
 * down updates to the local Room database
 */
viewModel.startCloudSync(currentUserId);
```

```
        layout.findViewById(R.id.btnPositive).setOnClickListener( View v -> {
            // Clear the SharedPreferences session on logout
            getSharedPreferences( name: "EventPrefs", MODE_PRIVATE).edit().clear().apply();

            /** SECURITY FEATURE: Clear Local Room Cache on Logout
            * Use a background thread to delete the user. The @ForeignKey CASCADE
            * rule setup in Event.java, will help to delete all their local events
            */
            AppDatabase.databaseWriteExecutor.execute(() -> {
                AppDatabase db = AppDatabase.getInstance( context: this);
                db.eventDao().deleteAllForUser(currentUserId); // Deletes local events
                db.userDao().deleteUserById(currentUserId);   // Deletes local user
            });

            Intent intent = new Intent( packageContext: this, LoginActivity.class);
            intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
            startActivity(intent);
            finish();
        });
        layout.findViewById(R.id.btnNegative).setOnClickListener( View v -> dialog.dismiss());
        dialog.show();
    }
```

Testing and Fixing Authentication Issues

During the initial testing of the enhanced application, I identified two minor issues within the authentication workflow. First, although the system correctly enforced a lockout after five consecutive failed login attempts, the failedAttempts counter was not reset in either the local or cloud database after the cooldown period. As a result, even when the lockout duration had elapsed, users were unable to log in with valid credentials. Second, during password recovery

testing, the MFA code was not received. This occurred because the application had been reinstalled for a clean test environment, and the user had not previously granted SMS permissions. Consequently, although the MFA code was generated, it was never delivered, leaving the user unable to proceed with authentication.



To resolve these issues, I implemented a validation (sanity check) prior to cloud authentication that evaluates the lockoutTimestamp. If the cooldown period has expired, the system automatically resets the failedAttempts counter, restoring normal login functionality. Additionally, I introduced a wrapper method to verify SMS permissions during both login and password recovery flows. If the required permissions are not granted, the application prompts the user to authorize them before attempting to send the MFA code, ensuring reliable delivery and preventing user lockout.

```
// Perform security checks safely on a background thread
AppDatabase.databaseWriteExecutor.execute(() -> {
    long now = System.currentTimeMillis();

    // if the penalty time has expired, reset the brute-force counters
    if (cloudUser.lockoutTimestamp > 0 && now > cloudUser.lockoutTimestamp) {
        cloudUser.failedAttempts = 0;
        cloudUser.lockoutTimestamp = 0;
    }

    // Check Brute-Force Attacks and implements a cooldown
    if (cloudUser.lockoutTimestamp > now) {
        long remaining = (cloudUser.lockoutTimestamp - now) / 60000;
        listener.onFinished( user: null,  statusMessage: "Account locked. Try again in " + (remainin
        return;
    }
```

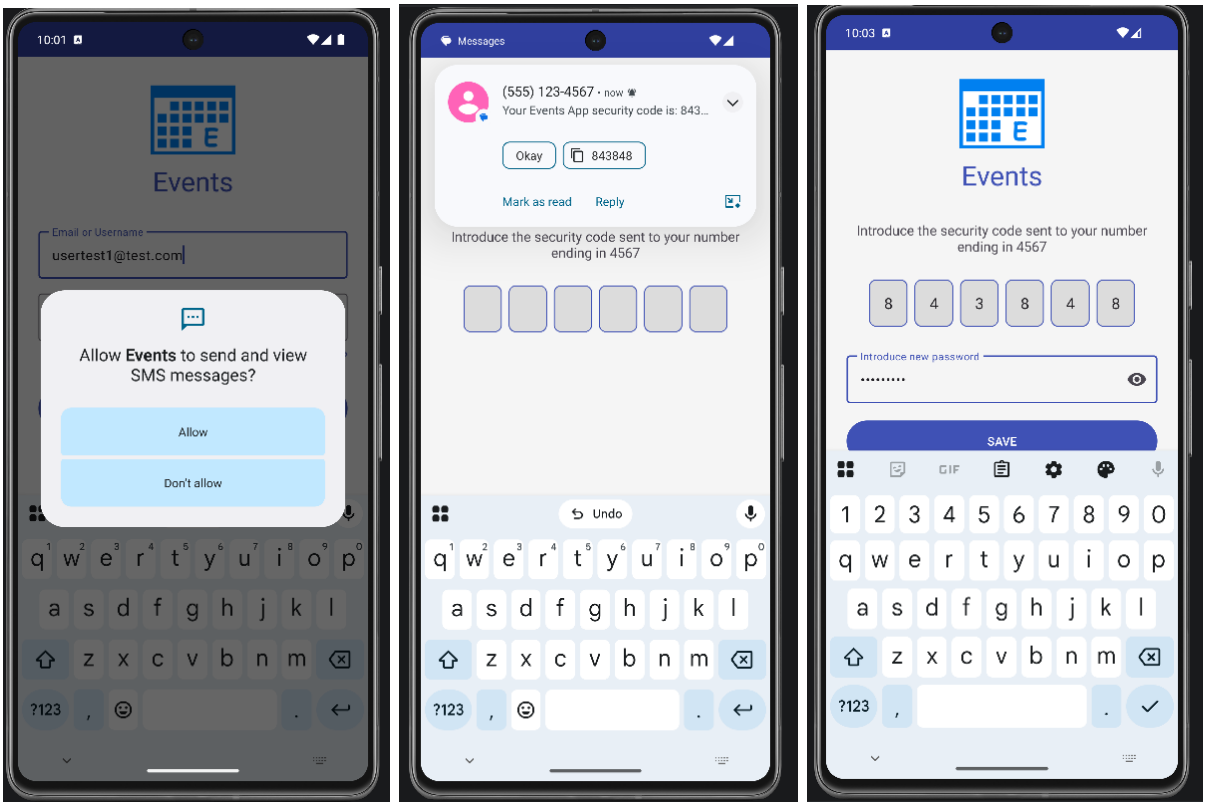
```java
// Helper method to ensure that users have permission before proceeding to SMS MFA flows
2 usages
private void executeWithSmsPermission(Runnable action) {
    if (ContextCompat.checkSelfPermission( context: this, Manifest.permission.SEND_SMS) == PackageManager.PERMISSION_GRANTED) {
        action.run(); // Permission already exists, run immediately
    } else {
        pendingAuthAction = action; // Save what the user was trying to do
        ActivityCompat.requestPermissions( activity: this, new String[]{Manifest.permission.SEND_SMS}, SMS_PERMISSION_CODE);
    }
}

// Prompt a dialog to ask user for SMS permission to enable MFA
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == SMS_PERMISSION_CODE) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            // User clicked "Allow". Automatically run the login/recovery action they initiated!
            if (pendingAuthAction != null) {
                pendingAuthAction.run();
                pendingAuthAction = null;
            }
        } else {
            Toast.makeText( context: this, text: "SMS Permission is required to receive login security codes.", Toast.LENGTH_LONG).s
        }
    }
}
```

```java
// Log In
btnLogin.setOnClickListener( View v -> {
    executeWithSmsPermission(() -> {
        viewModel.login(editEmailOrUser.getText().toString(), editPassword.getText().toString());
    });
});

// Forgot Password
txtForgotPassword.setOnClickListener( View v -> {
    executeWithSmsPermission(() -> {
        viewModel.startPasswordRecovery(editEmailOrUser.getText().toString());
    });
});
```

Cloud Implementation Testing

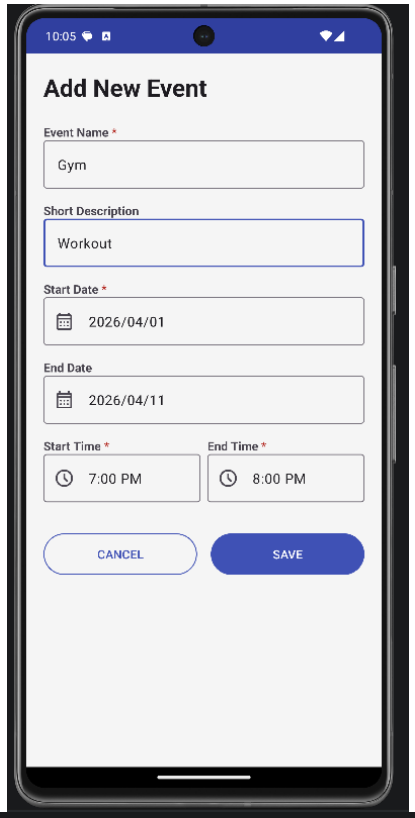
- MFA for Password Recovery (No SMS Permission Granted):

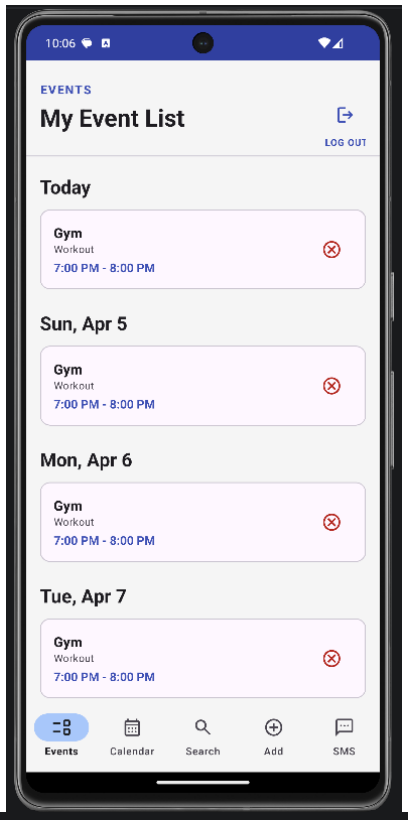
Validates that when SMS permissions are not granted, the system prompts the user to authorize them before attempting to send the MFA code, ensuring the recovery process can proceed successfully.

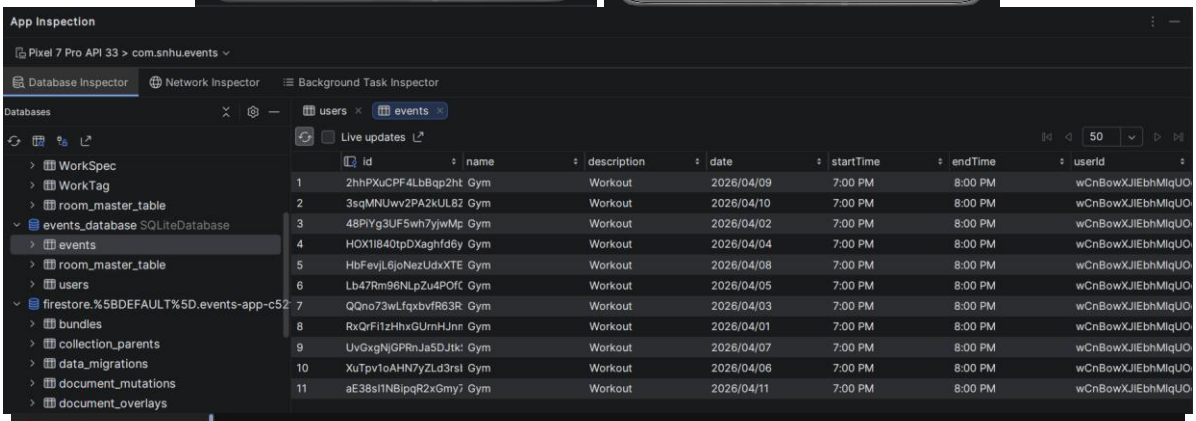


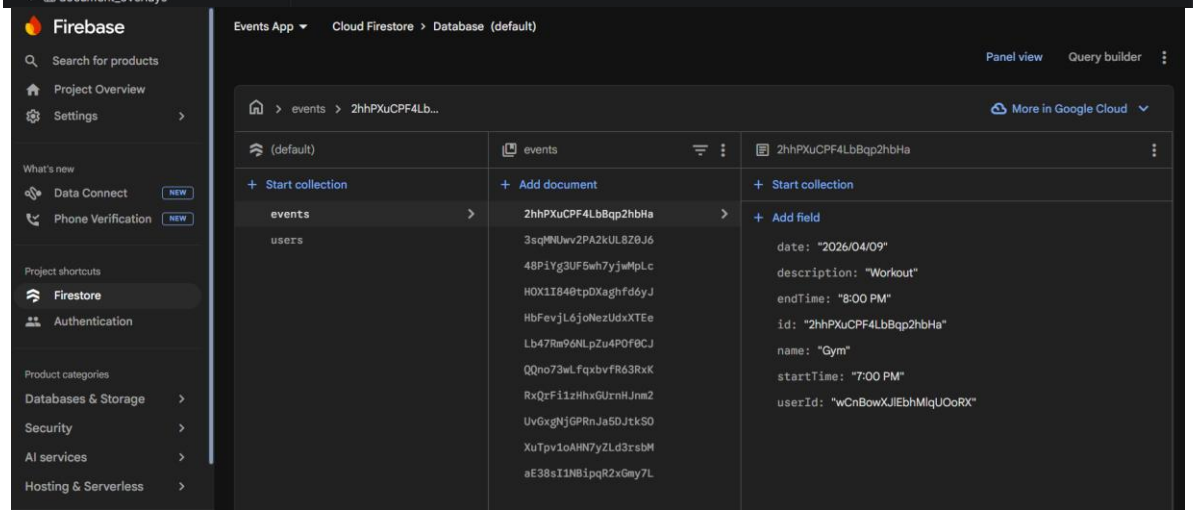
- Create New Events:

Verifies that newly created events are correctly displayed on the UI, stored in the local Room database, and synchronized with the Firestore cloud database.


10:05
Add New Event
Event Name *
Gym
Short Description
Workout
Start Date *
2026/04/01
End Date
2026/04/11
Start Time *
7:00 PM
End Time *
8:00 PM
CANCEL
SAVE

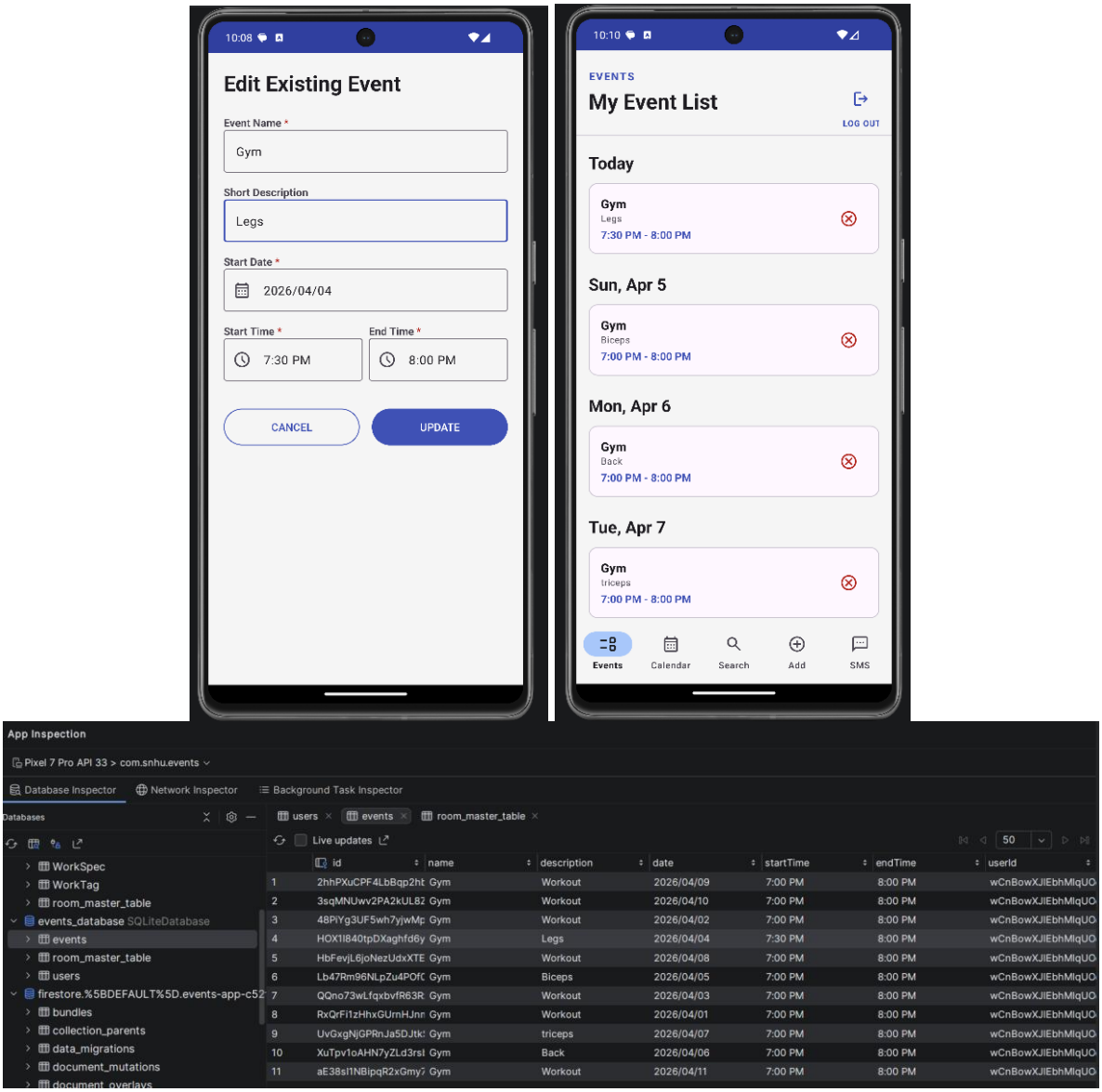

10:06
EVENTS
My Event List
LOG OUT
Today
Gym
Workout
7:00 PM - 8:00 PM
Sun, Apr 5
Gym
Workout
7:00 PM - 8:00 PM
Mon, Apr 6
Gym
Workout
7:00 PM - 8:00 PM
Tue, Apr 7
Gym
Workout
7:00 PM - 8:00 PM
Events
Calendar
Search
Add
SMS

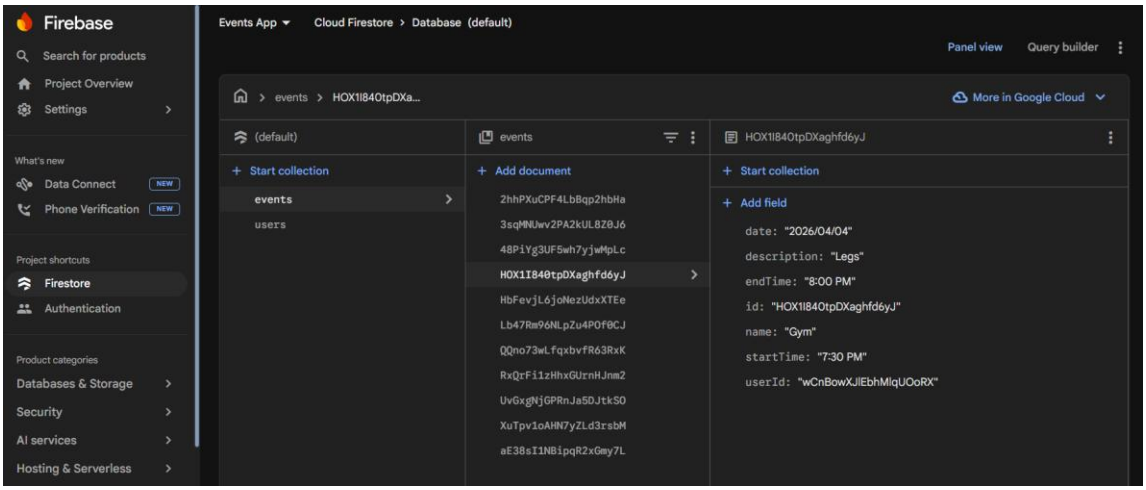

App Inspection
Pixel 7 Pro API 33 > com.snhu.events
Database Inspector
Network Inspector
Background Task Inspector
Databases
users
events
Live updates
WorkSpec
WorkTag
room_master_table
events_database SQLiteDatabase
events
room_master_table
users
firestore.%5BDEFAULT%5D.events-app-c52
bundles
collection_parents
data_migrations
document_mutations
document_overlays
id
name
description
date
startTime
endTime
userId
1 2hhPXuCPF4LbBqp2ht Gym Workout 2026/04/09 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
2 3sqMNUwv2PA2kUL8Z Gym Workout 2026/04/10 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
3 48PiYg3UF5wh7yjwMp Gym Workout 2026/04/02 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
4 HOX1I840tpDXaghfd6y Gym Workout 2026/04/04 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
5 HbFevjL6joNezUdxXTE Gym Workout 2026/04/08 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
6 Lb47Rm96NLpZu4POfC Gym Workout 2026/04/05 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
7 QQno73wLfqxbvfR63R Gym Workout 2026/04/03 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
8 RxQrFi1zHhxGUrnHJnn Gym Workout 2026/04/01 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
9 UvGxgNjGPRnJa5DJtk Gym Workout 2026/04/07 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
10 XuTpv1oAHN7yZLd3rsl Gym Workout 2026/04/06 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO
11 aE38sI1NBipqR2xGmy7 Gym Workout 2026/04/11 7:00 PM 8:00 PM wCnBowXJlEbhMlqUO


Firebase
Search for products
Project Overview
Settings
What's new
Data Connect
Phone Verification
Project shortcuts
Firestore
Authentication
Product categories
Databases & Storage
Security
AI services
Hosting & Serverless
Events App
Cloud Firestore > Database (default)
Panel view
Query builder
events > 2hhPXuCPF4Lb...
More in Google Cloud
(default)
events
2hhPXuCPF4LbBqp2hbHa
Start collection
Add document
Add field
events
users
2hhPXuCPF4LbBqp2hbHa
3sqMNUwv2PA2kUL8Z0J6
48PiYg3UF5wh7yjwMpLc
HOX1I840tpDXaghfd6yJ
HbFevjL6joNezUdxXTEe
Lb47Rm96NLpZu4POf0CJ
QQno73wLfqxbvfR63RxK
RxQrFi1zHhxGUrnHJnm2
UvGxgNjGPRnJa5DJtkSO
XuTpv1oAHN7yZLd3rsbM
aE38sI1NBipqR2xGmy7L
date: "2026/04/09"
description: "Workout"
endTime: "8:00 PM"
id: "2hhPXuCPF4LbBqp2hbHa"
name: "Gym"
startTime: "7:00 PM"
userId: "wCnBowXJlEbhMlqUOoRX"

- Edit Existing Events:

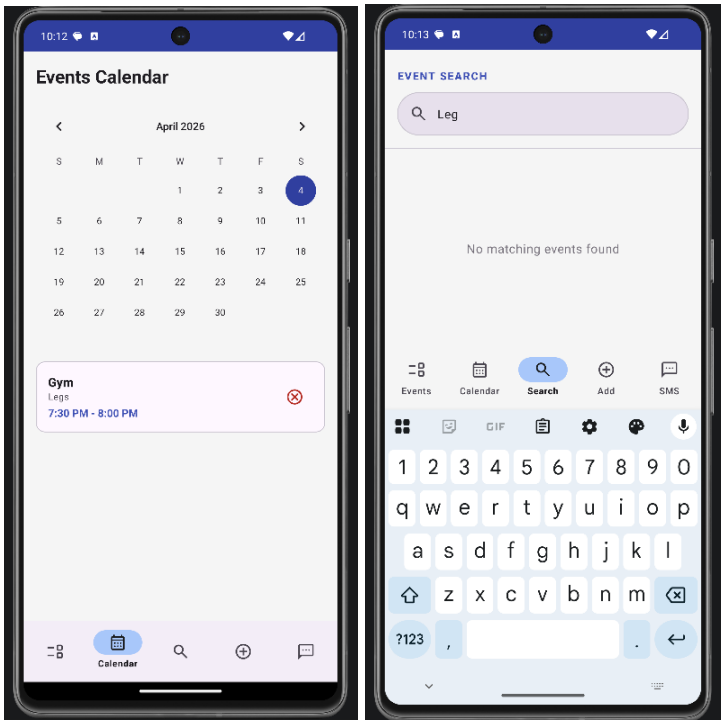
Confirms that updates to a selected event are accurately reflected on the UI, persisted in the local Room database, and synchronized with the cloud master database.

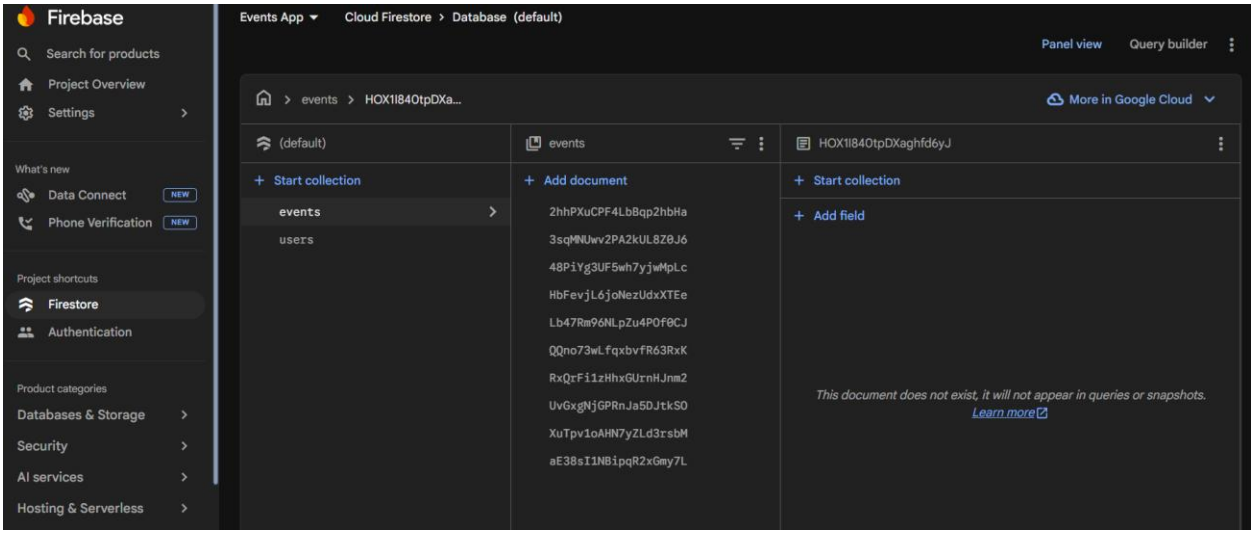




- Delete Events and Calendar Validation:

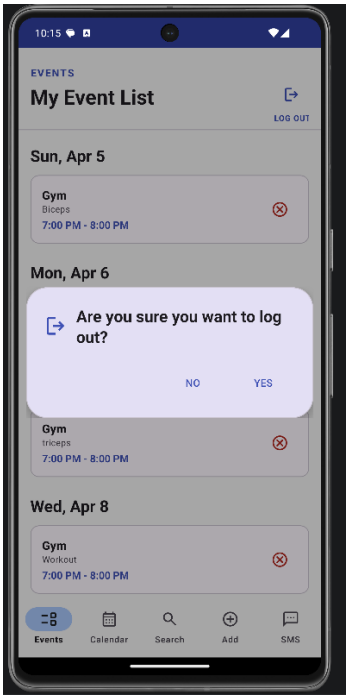
Ensures that deleting an event from the calendar view removes it from the UI, the local Room database, and the cloud database, maintaining data consistency across all layers.

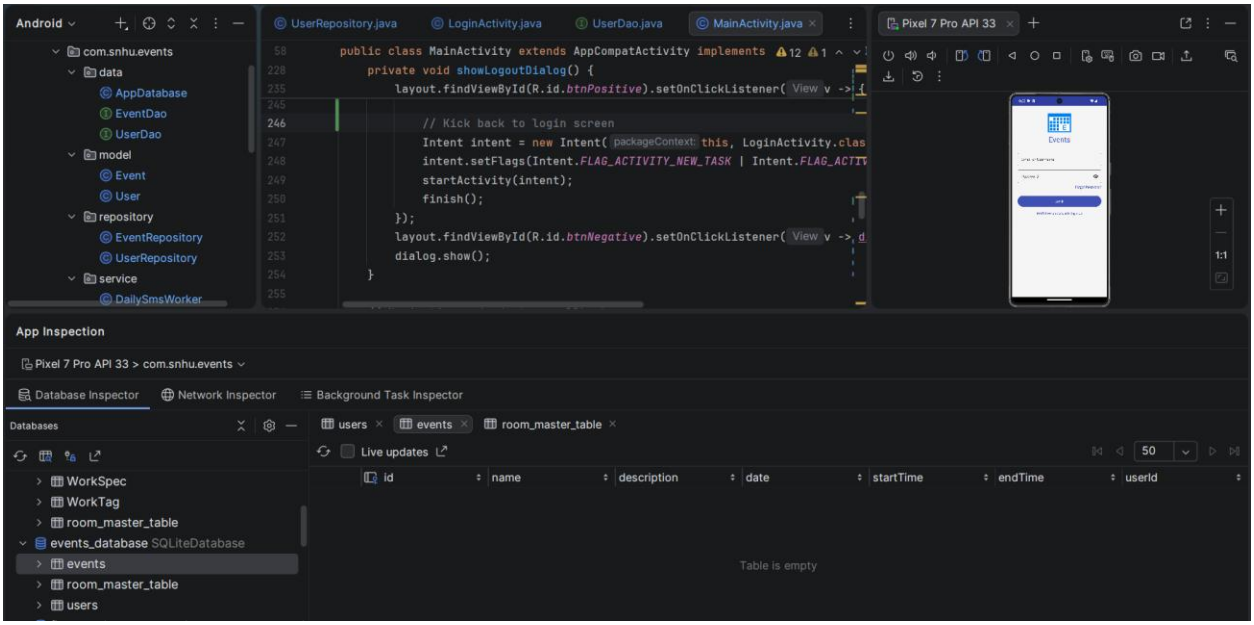


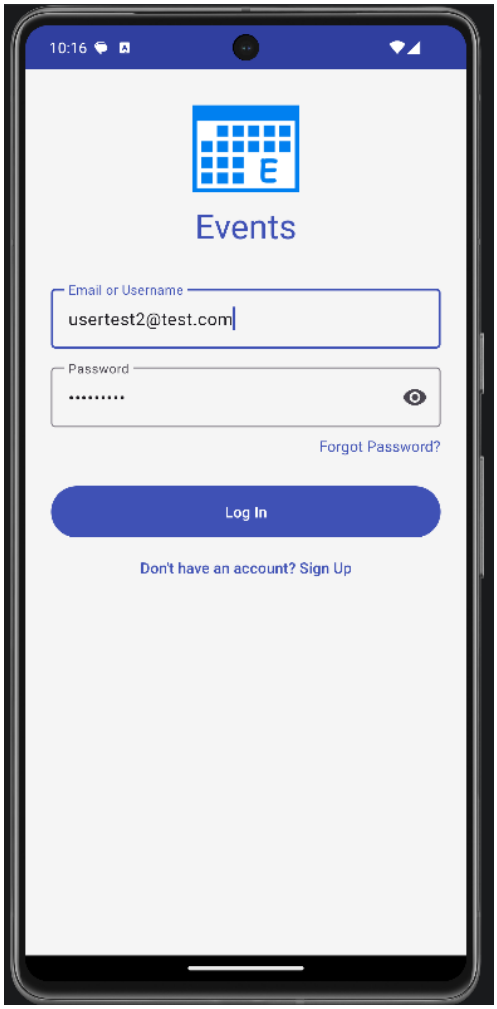


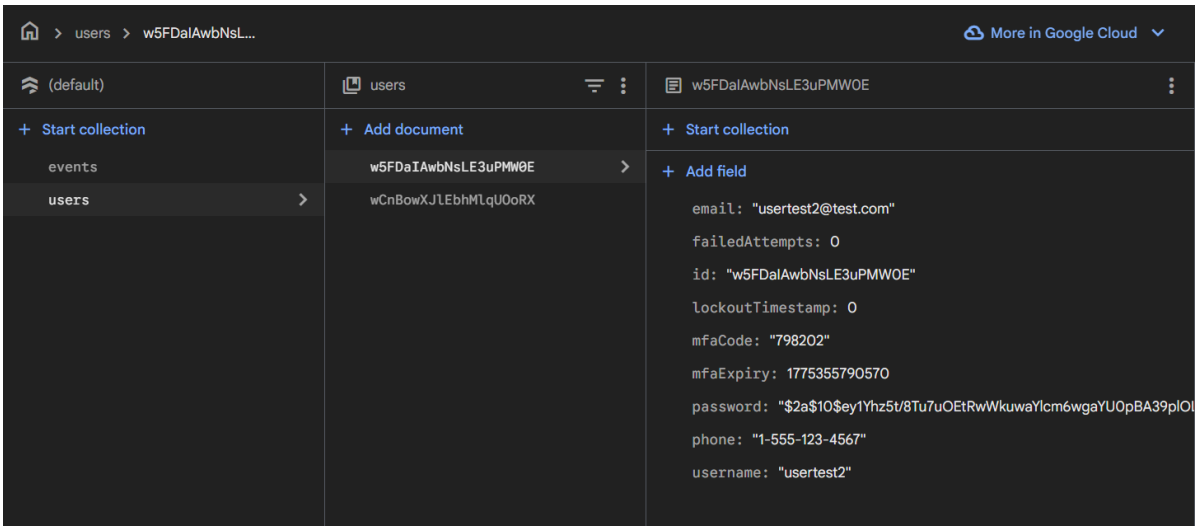
- Logout and Local Data Isolation:

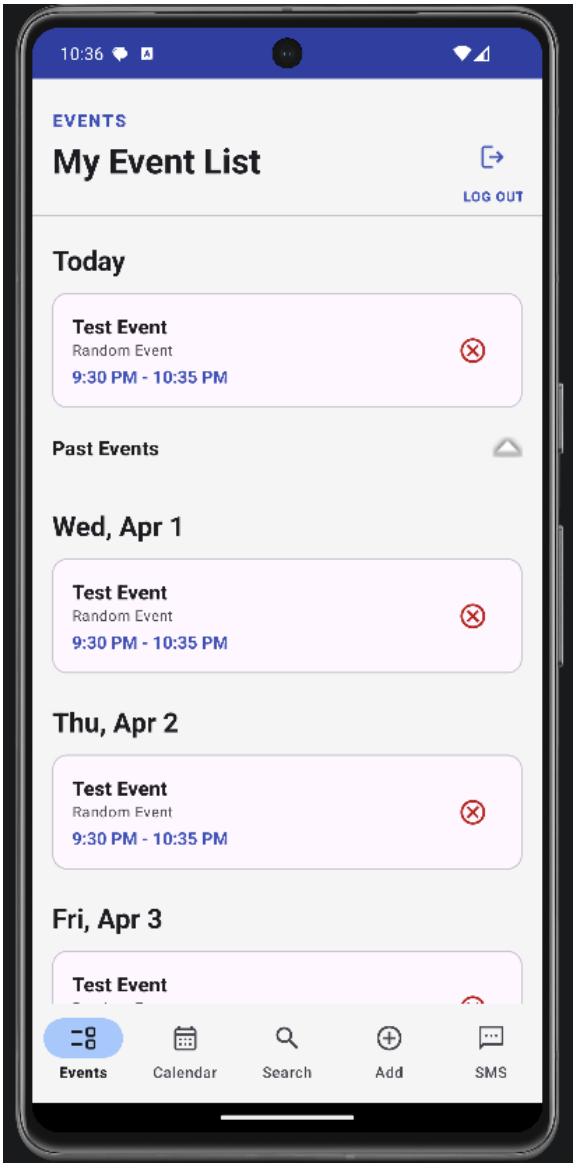
Validates that logging out clears all local Room database tables. When a new user is registered, their data is correctly stored both locally and in the cloud; however, the local database only retains data for the currently authenticated user, while the cloud database maintains records for all users.

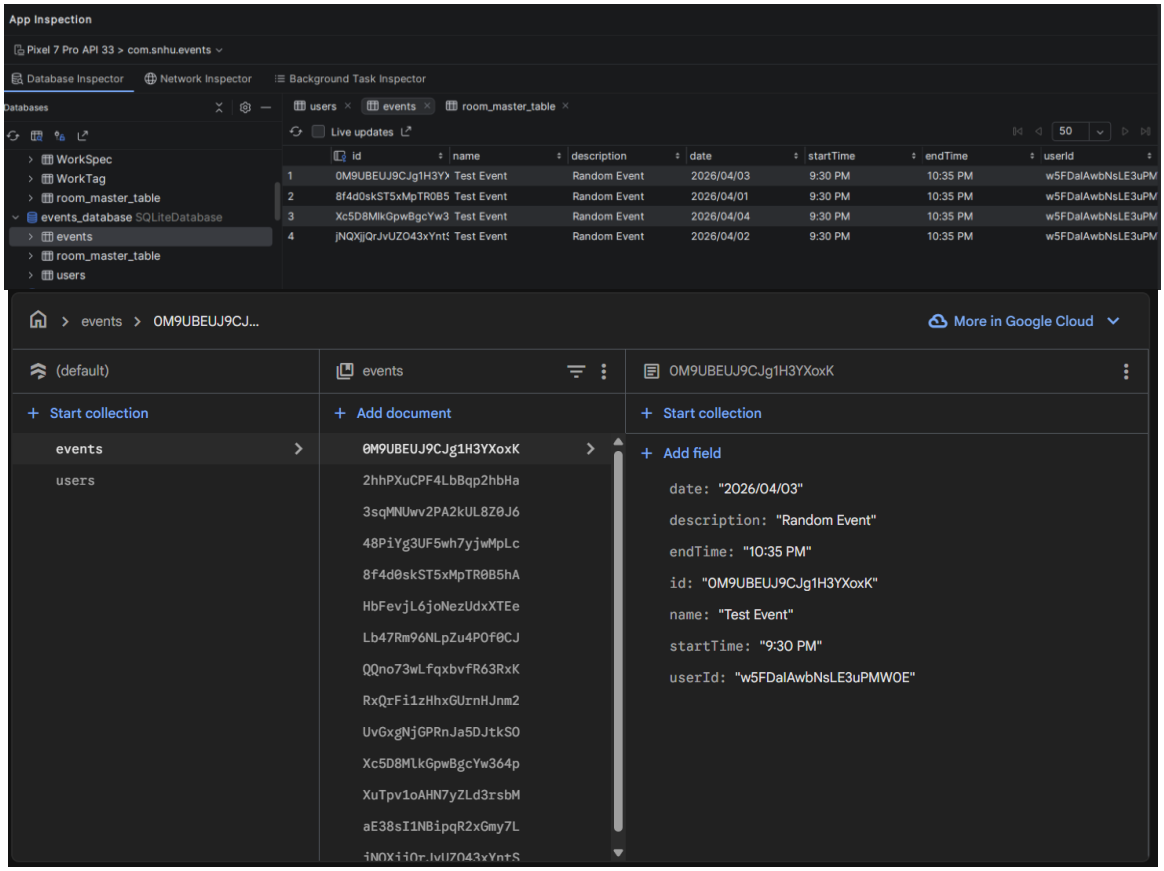



Android
com.snhu.events
data
AppDatabase
EventDao
UserDao
model
Event
User
repository
EventRepository
UserRepository
service
DailySmsWorker
UserRepository.java
LoginActivity.java
UserDao.java
MainActivity.java
public class MainActivity extends AppCompatActivity implements
private void showLogoutDialog() {
layout.findViewById(R.id.btnPositive).setOnClickListener(
// Kick back to login screen
Intent intent = new Intent(
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTI
startActivity(intent);
finish();
});
layout.findViewById(R.id.btnNegative).setOnClickListener(
dialog.show();
}
Pixel 7 Pro API 33
App Inspection
Pixel 7 Pro API 33 > com.snhu.events
Database Inspector
Network Inspector
Background Task Inspector
Databases
WorkSpec
WorkTag
room_master_table
events_database SQLiteDatabase
events
room_master_table
users
users
events
room_master_table
Live updates
id
name
description
date
startTime
endTime
userId
Table is empty


10:16
Events
Email or Username
usertest2@test.com
Password
Forgot Password?
Log In
Don't have an account? Sign Up

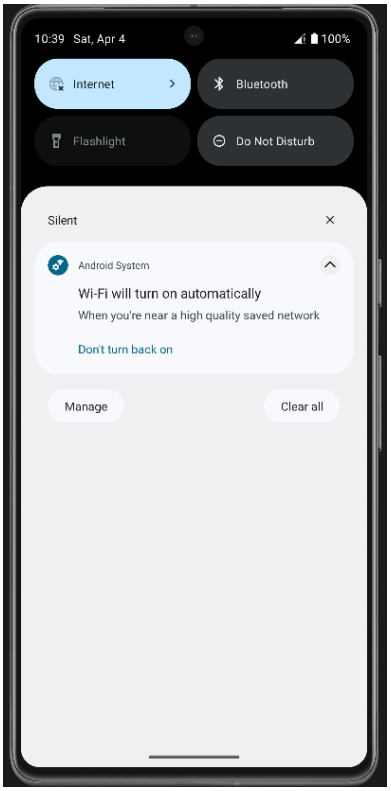

users > w5FDaIAwbNsL...
More in Google Cloud
(default)
users
w5FDaIAwbNsLE3uPMW0E
+ Start collection
events
users
+ Add document
w5FDaIAwbNsLE3uPMW0E
wCnBowXJlEbhMlqUOoRX
+ Start collection
+ Add field
email: "usertest2@test.com"
failedAttempts: 0
id: "w5FDaIAwbNsLE3uPMW0E"
lockoutTimestamp: 0
mfaCode: "798202"
mfaExpiry: 1775355790570
password: "$2a$10$ey1Yhz5t/8Tu7uOEtRwWkuwaYlcm6wgaYU0pBA39plO
phone: "1-555-123-4567"
username: "usertest2"

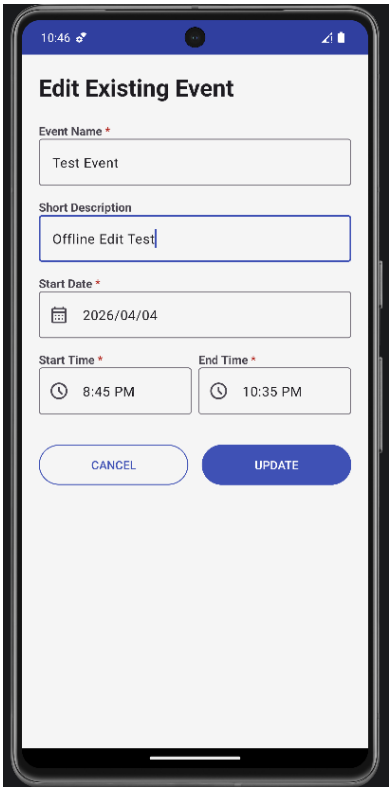

10:36
EVENTS
My Event List
LOG OUT
Today
Test Event
Random Event
9:30 PM - 10:35 PM
Past Events
Wed, Apr 1
Test Event
Random Event
9:30 PM - 10:35 PM
Thu, Apr 2
Test Event
Random Event
9:30 PM - 10:35 PM
Fri, Apr 3
Test Event
Events
Calendar
Search
Add
SMS

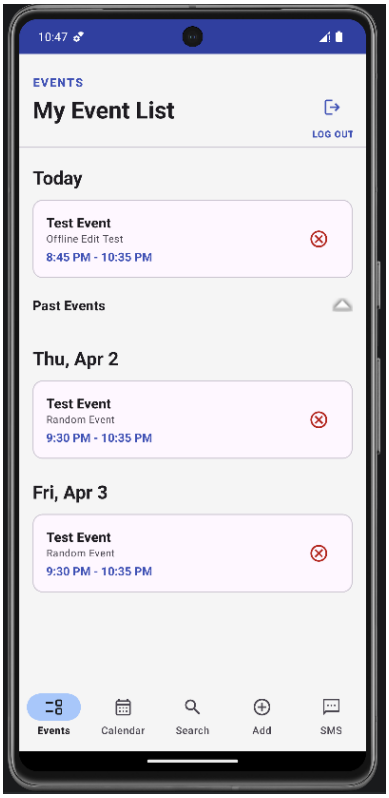


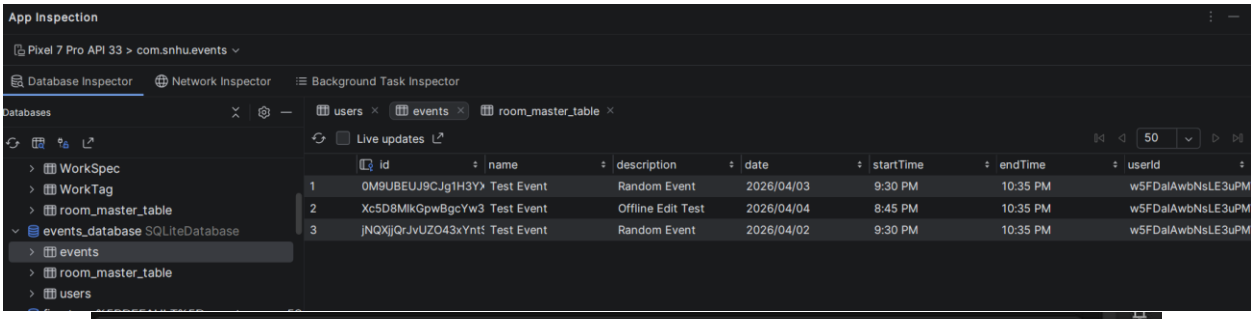
- Offline Mode (Create and Edit Events):

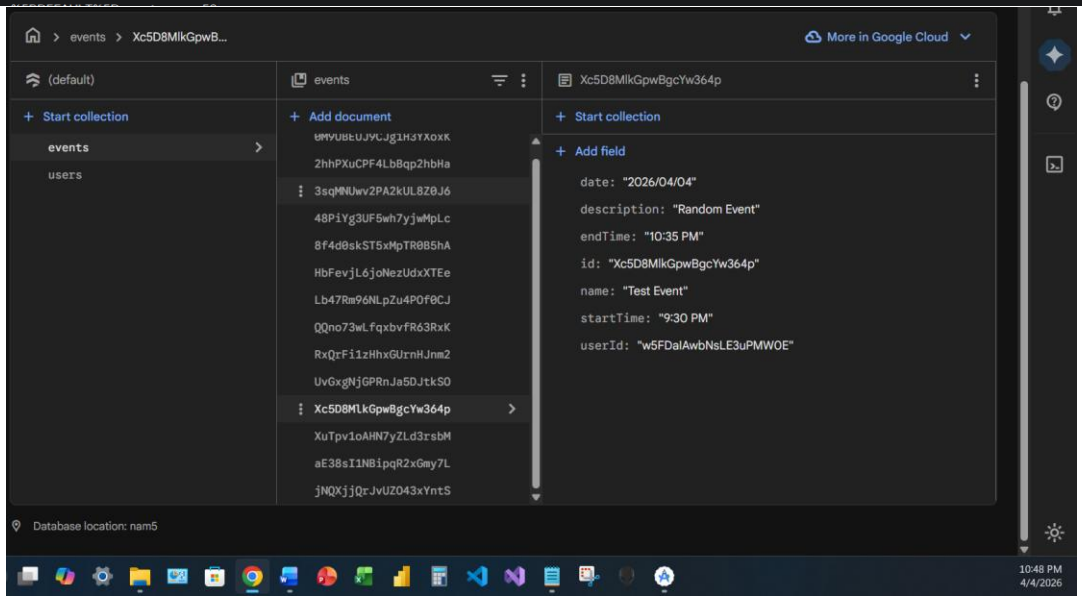
Tests that when the application is offline, newly created or edited events are stored locally in the Room database without affecting the cloud. Once connectivity is restored, the queued changes are automatically synchronized with the cloud database.

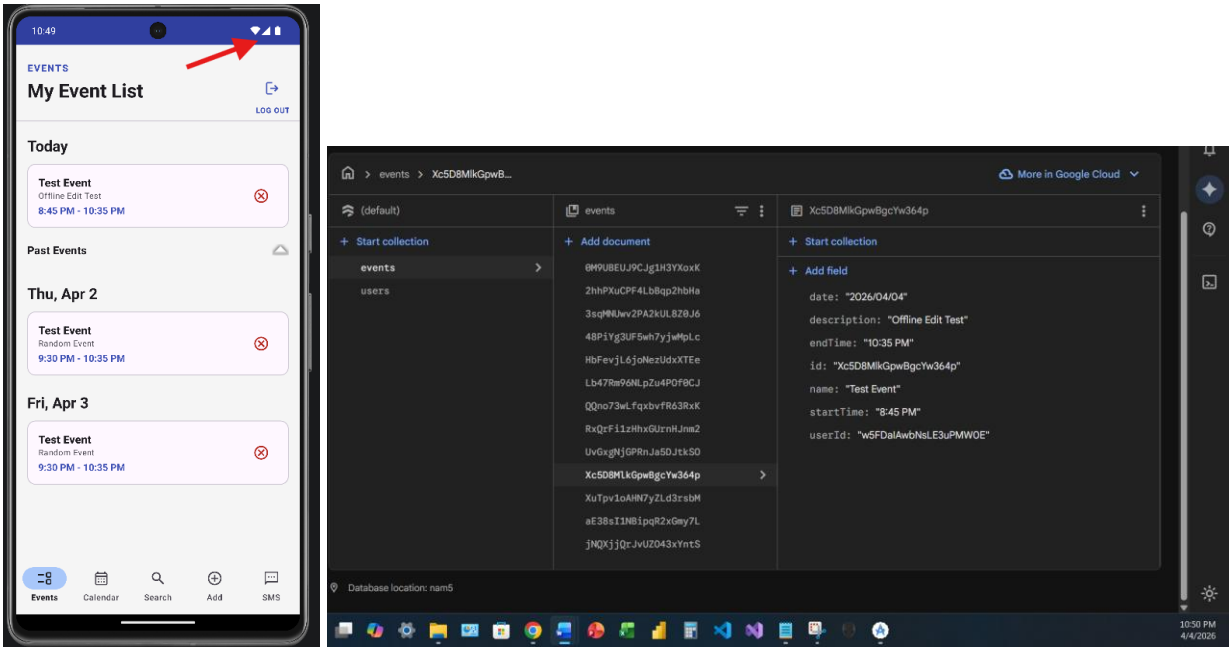

10:39 Sat, Apr 4
100%
Internet
Bluetooth
Flashlight
Do Not Disturb
Silent
Android System
Wi-Fi will turn on automatically
When you're near a high quality saved network
Don't turn back on
Manage
Clear all


10:46
Edit Existing Event
Event Name *
Test Event
Short Description
Offline Edit Test
Start Date *
2026/04/04
Start Time *
8:45 PM
End Time *
10:35 PM
CANCEL
UPDATE


10:47
EVENTS
My Event List
LOG OUT
Today
Test Event
Offline Edit Test
8:45 PM - 10:35 PM
Past Events
Thu, Apr 2
Test Event
Random Event
9:30 PM - 10:35 PM
Fri, Apr 3
Test Event
Random Event
9:30 PM - 10:35 PM
Events
Calendar
Search
Add
SMS

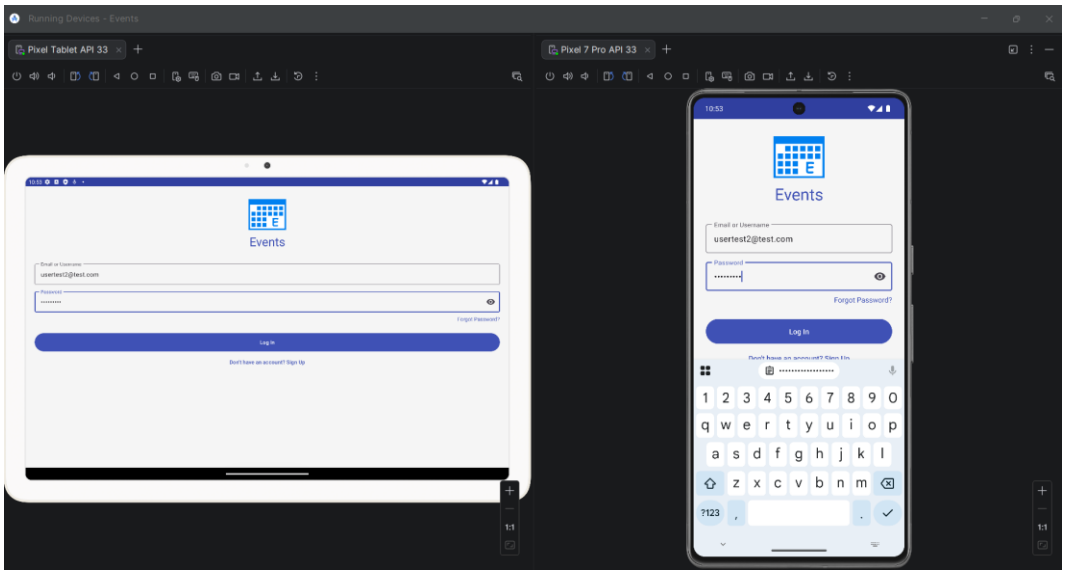

App Inspection
Pixel 7 Pro API 33 > com.snhu.events
Database Inspector
Network Inspector
Background Task Inspector
Databases
WorkSpec
WorkTag
room_master_table
events_database SQLiteDatabase
events
room_master_table
users
users
events
room_master_table
Live updates
50
id
name
description
date
startTime
endTime
userId
1 0M9UBEUJ9CJg1H3Y Test Event Random Event 2026/04/03 9:30 PM 10:35 PM w5FDalAwbNsLE3uPM
2 Xc5D8MlkGpwBgcYw3 Test Event Offline Edit Test 2026/04/04 8:45 PM 10:35 PM w5FDalAwbNsLE3uPM
3 jNQXjjQrJvUZO43xYnt Test Event Random Event 2026/04/02 9:30 PM 10:35 PM w5FDalAwbNsLE3uPM

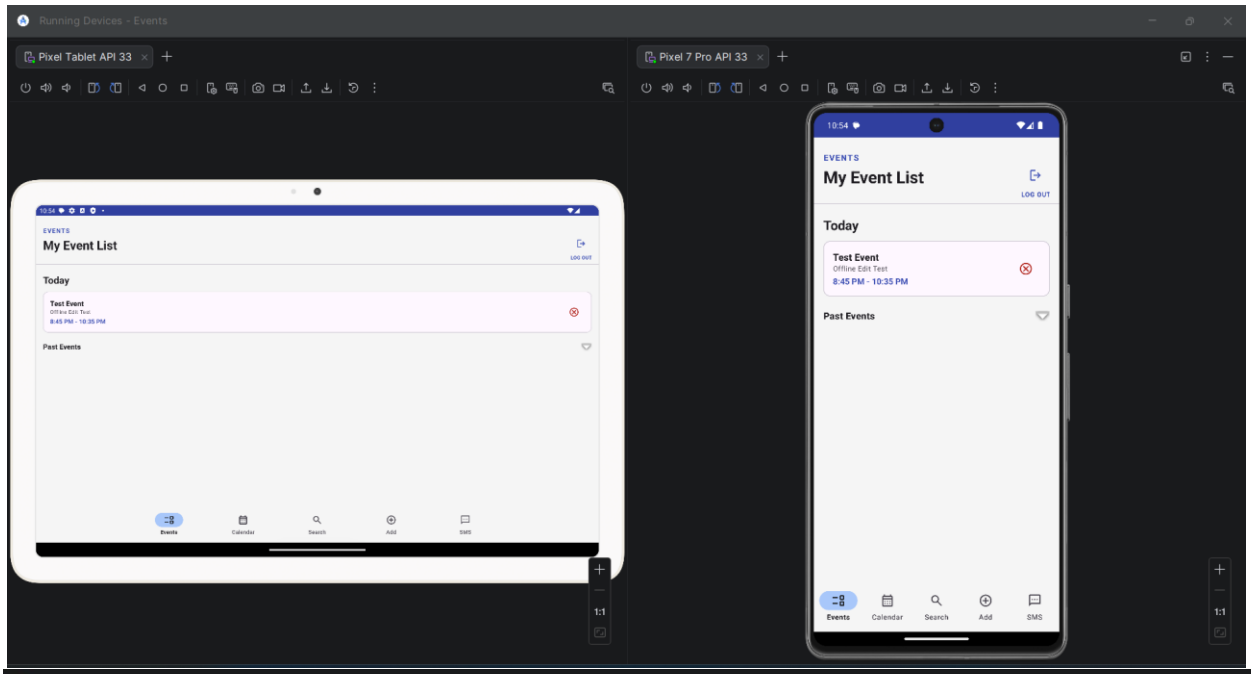

events > Xc5D8MlkGpwB...
More in Google Cloud
(default)
events
Xc5D8MlkGpwBgcYw364p
Start collection
events
users
Add document
2hhPXuCPF4LbBqp2hbHa
3sqMNUwv2PA2kUL8Z0J6
48PiYg3UF5wh7yjwMpLc
8f4d0skST5xMpTR0B5hA
HbFevjL6joNezUdxXTEe
Lb47Rm96NLpZu4POf0CJ
QQno73wLfqxbvfR63RxK
RxQrFi1zHhxGUrnHJnm2
UvGxgNjGPRnJa5DJtkSO
Xc5D8MlkGpwBgcYw364p
XuTpv1oAHN7yZLd3rsbM
aE38sI1N8ipqR2xGmy7L
jNQXjjQrJvUZO43xYntS
Start collection
Add field
date: "2026/04/04"
description: "Random Event"
endTime: "10:35 PM"
id: "Xc5D8MlkGpwBgcYw364p"
name: "Test Event"
startTime: "9:30 PM"
userId: "w5FDalAwbNsLE3uPMWOE"
Database location: nam5
10:48 PM
4/4/2026

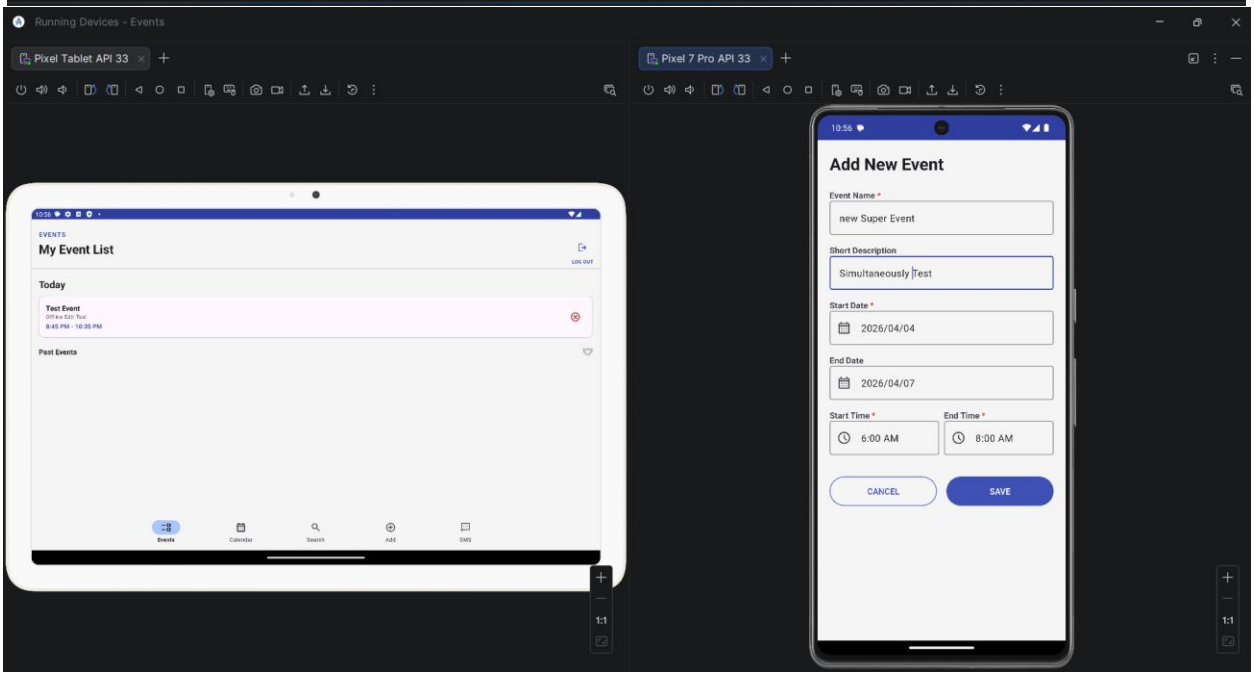


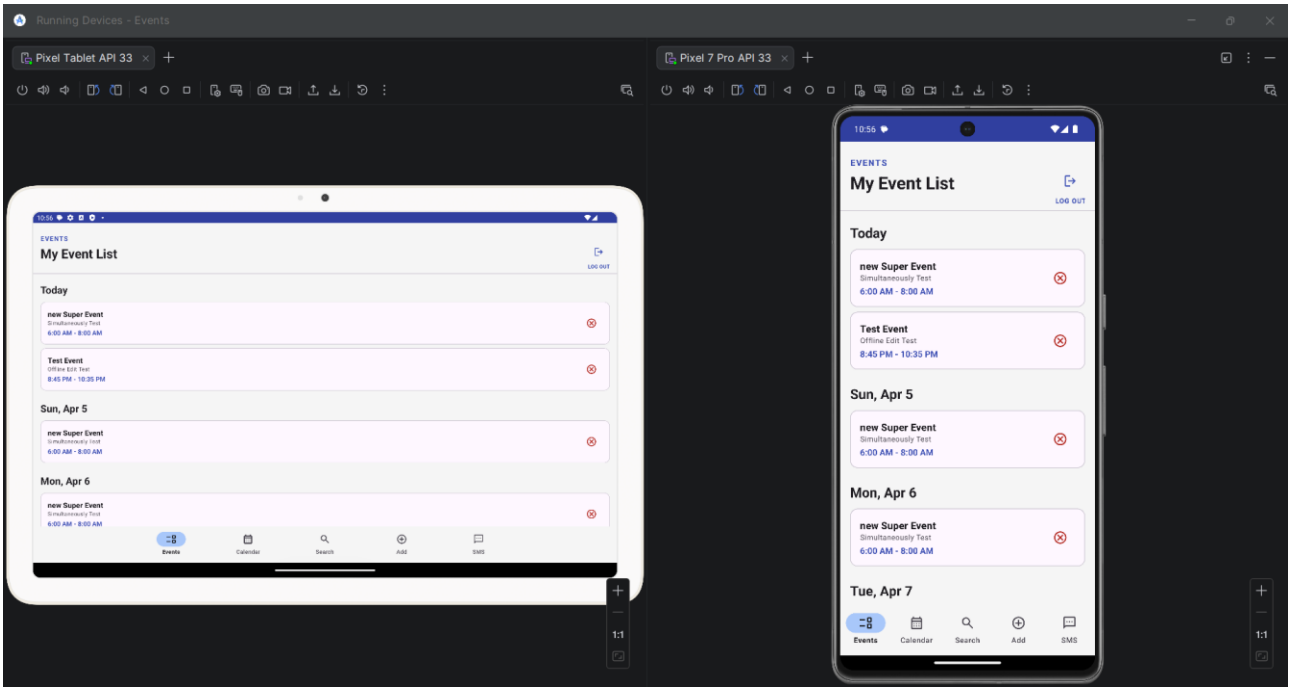
- Multi-Device Synchronization:

Simulates concurrent sessions on multiple devices (e.g., tablet and mobile phone) using the same user account. Verifies that all events are consistently displayed across devices and that newly created events on one device are immediately reflected on the other through real-time synchronization.




Running Devices - Events
Pixel Tablet API 33
Pixel 7 Pro API 33
EVENTS
My Event List
LOG OUT
Today
Test Event
Offline Edit Test
8:45 PM - 10:35 PM
Past Events
Events
Calendar
Search
Add
SMS


Running Devices - Events
Pixel Tablet API 33
Pixel 7 Pro API 33
EVENTS
My Event List
LOG OUT
Today
Test Event
Past Events
Add New Event
Event Name *
new Super Event
Short Description
Simultaneously Test
Start Date *
2026/04/04
End Date
2026/04/07
Start Time *
6:00 AM
End Time *
8:00 AM
CANCEL
SAVE



Conclusions and Course Outcomes Alignment

Through this database enhancement, I developed a comprehensive understanding of modern data architecture by transitioning from a local-only relational model to a hybrid cloud-based solution. By implementing an offline-first approach combined with a cloud-authoritative synchronization pattern, I was able to balance performance, scalability, and reliability. Leveraging Firebase Firestore as the master source of truth and Room as a local caching layer allowed the application to maintain responsiveness while ensuring cross-device data consistency. Throughout this process, I applied key software engineering principles such as separation of concerns (MVVM), repository pattern abstraction, conflict resolution strategies, and efficient dependency management. Additionally, I incorporated best practices in security, including user-scoped data isolation, controlled authentication queries, MFA integration, and permission handling, while also addressing real-world challenges such as dependency conflicts, asynchronous data flows, and offline synchronization.

These enhancements strongly align with the five course outcomes. I employed collaborative and documentation strategies using version control and structured implementation workflows, supporting Outcome 1. I communicated complex architectural decisions and technical implementations clearly through structured documentation and testing scenarios, aligning with Outcome 2. The design and evaluation of a hybrid cloud architecture, including trade-off analysis between local and distributed systems, demonstrates Outcome 3. Furthermore, the use of modern tools and techniques, such as Firestore, Room, real-time listeners, batch operations, and offline persistence, reflects Outcome 4 by delivering a scalable and industry-relevant solution. Finally, I developed a strong security mindset aligned with Outcome 5 by anticipating potential vulnerabilities, enforcing data access restrictions, and implementing secure authentication and synchronization mechanisms. Overall, this enhancement not only improved the application's functionality but also strengthened my ability to design and implement robust, secure, and scalable data-driven systems.

References

Amplify Hosting. (2026, March 26). Amazon Web Services, Inc. https://aws.amazon.com/amplify/hosting/?nc=sn&amp

Build an offline-first app. (n.d.). Android Developers. https://developer.android.com/topic/architecture/data-layer/offline-first

Enabling Offline Capabilities on Android | Firebase Realtime Database. (n.d.). Firebase. https://firebase.google.com/docs/database/android/offline-capabilities

Firebase. (2019). *Firebase Pricing | Firebase*. Firebase. https://firebase.google.com/pricing

Firebase Android Release Notes. (n.d.). Firebase. https://firebase.google.com/support/release-notes/android

Google. (n.d.). *Get realtime updates with Cloud Firestore*. Firebase. https://firebase.google.com/docs/firestore/query-data/listen

MongoDB. (n.d.). *Pricing*. MongoDB. https://www.mongodb.com/pricing

Sonar. (2019, March 19). *Dependency hell: a complete guide*. Sonarsource.com; Sonar. https://www.sonarsource.com/blog/dependency-hell

Team, M. D. (2026). *Monitor Data Changes*. Mongodb.com. https://www.mongodb.com/docs/languages/java/reactive-streams-driver/current/read/change-streams/