# Firebase Audio File Upload in Android Kotlin: Cloud Storage & Firestore Metadata Integration

## 1. Introduction

Firebase Cloud Storage provides a scalable solution for storing and serving user-generated content, such as **MIDI** and **WAV** files.

This report presents a comprehensive walkthrough for integrating Firebase Authentication, Cloud Storage, and Firestore in an Android app using Kotlin.

We’ll cover best practices for security, resumable uploads, progress monitoring, metadata tracking, and testing with the Local Emulator Suite.

## 2. Project Setup and Dependencies

### 2.1 Adding Firebase Libraries

1. Open your module-level Gradle file (<project>/<app-module>/build.gradle.kts).
2. Import the Firebase Android BoM to manage versions consistently.
3. Add dependencies for Storage, Firestore, Authentication, and Coroutines support.

|  |
| --- |
| dependencies {  // Firebase BoM for version alignment  implementation(platform("com.google.firebase:firebase-bom:34.0.0"))   // Core Firebase libraries  implementation("com.google.firebase:firebase-auth-ktx") // Authentication  implementation("com.google.firebase:firebase-storage-ktx") // Cloud Storage  implementation("com.google.firebase:firebase-firestore-ktx") // Firestore   // Kotlin Coroutines support for Tasks  implementation("org.jetbrains.kotlinx:kotlinx-coroutines-play-services:1.6.4") } |

By using KTX libraries and Coroutines extensions, you gain **await()** functions for cleaner asynchronous code 1.

### 2.2 Firebase Initialization

In your Application or MainActivity, initialize Firebase:

|  |
| --- |
| class MyApp : Application() {  override fun onCreate() {  super.onCreate()  FirebaseApp.initializeApp(this)  } } |

This ensures that Storage, Auth, and Firestore references can be accessed globally.

## 3. Firebase Authentication Integration

### 3.1 Enabling Authentication Providers

1. In the Firebase Console, navigate to **Authentication > Sign-in Method**.
2. Enable **Email/Password** (and any other providers as needed).

This setup is required before any Storage or Firestore operations by default .

### 3.2 In-App Authentication Code

|  |
| --- |
| // Initialize FirebaseAuth private val auth: FirebaseAuth = Firebase.auth  // Sign in existing users fun signIn(email: String, password: String, onResult: (Boolean, String?) -> Unit) {  auth.signInWithEmailAndPassword(email, password)  .addOnCompleteListener { task ->  if (task.isSuccessful) {  onResult(true, null)  } else {  onResult(false, task.exception?.localizedMessage)  }  } }  // Check current user state val currentUser = auth.currentUser |

Maintaining **authenticated state** is essential for enforcing security rules on Storage and Firestore .

## 4. Security Rules for Cloud Storage and Firestore

### 4.1 Cloud Storage Rules

|  |
| --- |
| rules\_version = '2'; service firebase.storage {  match /b/{bucket}/o {  // Restrict writes and reads to authenticated users  match /audio/{filePath=\*\*} {  allow read, write: if request.auth != null;  }  } } |

These rules ensure only logged-in users can upload or download files .

### 4.2 Firestore Security Rules

|  |
| --- |
| rules\_version = '2'; service cloud.firestore {  match /databases/{database}/documents {  // Allow users to create and read their own metadata  match /audioMetadata/{docId} {  allow create: if request.auth != null;  allow read: if request.auth != null && request.auth.uid == resource.data.userId;  }  } } |

Leveraging **user-based access** prevents unauthorized metadata access 2.

## 5. Uploading Audio Files to Cloud Storage

### 5.1 Preparing the File Reference

|  |
| --- |
| // Cloud Storage reference private val storage = Firebase.storage private fun getAudioRef(fileName: String): StorageReference =  storage.reference.child("audio/$fileName") |

Use **child()** to point to a logical path for audio assets .

### 5.2 Including Metadata

|  |
| --- |
| val metadata = storageMetadata {  contentType = "audio/wav" // MIME type for WAV files  setCustomMetadata("origin", "app-generated") // Custom key/value } |

Custom metadata helps track file origins or processing steps 3.

### 5.3 Upload Function with Progress Monitoring

|  |
| --- |
| suspend fun uploadAudioFile(  fileUri: Uri,  fileName: String ): String = withContext(Dispatchers.IO) {  val ref = getAudioRef(fileName)  val uploadTask = ref.putFile(fileUri, metadata)   // Monitor progress  uploadTask.snapshotEvents.collect { snapshot ->  val percent = (100.0 \* snapshot.bytesTransferred / snapshot.totalByteCount).toInt()  // Update UI or state: e.g., progressBar.progress = percent  }   // Await completion  uploadTask.await() // From kotlinx-coroutines-play-services  // Retrieve download URL  val downloadUrl = ref.downloadUrl.await()  return@withContext downloadUrl.toString() } |

* **snapshotEvents** for real-time progress updates
* **await()** for Task completion without callbacks4

### 5.4 Resumable Uploads

Continuing uploads after restarts relies on saving the session URI:

|  |
| --- |
| val sessionUri = uploadTask.snapshot.uploadSessionUri // Persist sessionUri (e.g., SharedPreferences) // On restart, call putFile(fileUri, metadata).resumeFrom(sessionUri) |

Sessions last one week, improving reliability for large files .

## 6. Firestore Metadata Tracking

### 6.1 Defining Data Model

|  |
| --- |
| data class AudioMetadata(  val fileName: String = "",  val downloadUrl: String = "",  val userId: String = "",  val timestamp: Long = System.currentTimeMillis() ) |

Keep documents small and selectively indexed for performance .

### 6.2 Writing Metadata Document

|  |
| --- |
| suspend fun saveMetadata(metadata: AudioMetadata) {  val db = Firebase.firestore  db.collection("audioMetadata")  .add(metadata)  .await() } |

Use **await()** for Firestore operations via Kotlin-Coroutines KTX 1.

## 7. Error Handling and Retry Strategies

### 7.1 Handling Common Errors

|  |
| --- |
| uploadTask.addOnFailureListener { exception ->  when ((exception as StorageException).errorCode) {  StorageException.ERROR\_NOT\_AUTHENTICATED -> /\* prompt user to log in \*/  StorageException.ERROR\_QUOTA\_EXCEEDED -> /\* inform about quota \*/  else -> /\* generic retry logic \*/  } } |

Refer to all Firebase Storage error codes for robust diagnosis 5.

### 7.2 Exponential Backoff

On transient failures, retry uploads after increasing delays:

|  |
| --- |
| var attempt = 0 while (attempt < MAX\_RETRIES) {  try {  uploadAudioFile(...)  break  } catch (e: Exception) {  delay(2.0.pow(attempt.toDouble()).toLong() \* 1000)  attempt++  } } |

This pattern reduces load during outages.

## 8. Lifecycle and Background Uploads

### 8.1 Activity Lifecycle Handling

Uploads continue even if Activities are destroyed; use **getActiveUploadTasks()** to reattach listeners .

### 8.2 WorkManager Integration

For guaranteed background execution:

|  |
| --- |
| class UploadWorker(...) : CoroutineWorker(...) {  override suspend fun doWork(): Result {  try {  val url = uploadAudioFile(inputData.getString("FILE\_URI")!!.toUri(), inputData.getString("NAME")!!)  saveMetadata(AudioMetadata(..., url))  return Result.success()  } catch (e: Exception) {  return Result.retry()  }  } } |

WorkManager ensures tasks run even if the app is closed 6.

## 9. Scalability and Naming Conventions

* Use **userId/timestamp\_filename.wav** paths to avoid hotspots.
* Choose a **multi-region bucket** for global low-latency access.
* Implement TTL policies in Firestore to expire old metadata .

A consistent naming scheme and regional considerations support high throughput with minimal contention.

## 10. Testing with Firebase Emulator Suite

1. Start emulators: firebase emulators:start --only auth,firestore,storage.
2. Point your app to local endpoints in debug builds.
3. Use the **Rules Playground** and integration tests to validate your security rules 7.

Testing locally avoids impacting production data and helps catch rule misconfigurations early.

## 11. Conclusion

This report provided an end-to-end solution for **uploading generated MIDI/WAV files** in Android with Kotlin.

We covered Authentication, secure Storage & Firestore Rules, resumable uploads, error handling, background execution, and local testing.

By following these patterns and best practices, your app will be **secure**, **scalable**, and **user-friendly**.

# References (7)

1. *firebase-android-sdk/docs/ktx/firestore.md at main - GitHub*. <https://github.com/firebase/firebase-android-sdk/blob/main/docs/ktx/firestore.md>

2. *Get started with Cloud Firestore Security Rules* . <https://firebase.google.com/docs/firestore/security/get-started>

3. *Adding Metadata to Uploaded Image Firebase/Kotlin*. <https://stackoverflow.com/questions/61581087/adding-metadata-to-uploaded-image-firebase-kotlin>

4. *Using Firebase with Kotlin coroutines - Stack Overflow*. <https://stackoverflow.com/questions/57562864/using-firebase-with-kotlin-coroutines>

5. *Handle errors for Cloud Storage on Android - Firebase*. <https://firebase.google.com/docs/storage/android/handle-errors>

6. *android - Firebase storage wait for listeners - Stack Overflow*. <https://stackoverflow.com/questions/43004365/firebase-storage-wait-for-listeners>

7. *Introduction to Firebase Local Emulator Suite*. <https://firebase.google.com/docs/emulator-suite>