













Mark Your Attendance — Project Overview (Using Firebase, GetX, and Clean Architecture)

This document outlines the setup, prompt strategy, manual tasks, and core features for building the **Mark Your Attendance** using **Flutter 3.27.2**, **Firebase**, **GetX**, and **Clean Architecture** principles. The project uses **Cursor AI** prompts to scaffold code and ensure scalable, modular architecture. It includes:

-  Features of the Application
-  Tasks performed manually
-  Tasks completed using **Cursor AI Prompts**
-  Additional configurations
-  Testing Checklist
-  Final Summary
- [Project GitHub link.](#)

Features of the Application

-  Phone number authentication using OTP
 -  Location tracking on sign-in/sign-up
 -  Internet connectivity check before key actions
 -  Auto-redirect to login if unauthenticated
 -  Real-time attendance logging (Planned)
 -  Dashboard for viewing attendance (Planned)
 -  Profile update including phone number change
-

✓ Manually Performed Tasks

🔧 Project & Environment Setup

- Flutter SDK: 3.27.2 (stable)
- Dart SDK: 3.6.1
- IDE: Cursor

Initialized project with:

- flutter create mark_your_attendance

🔑 Firebase Configuration

- Created a Firebase project at console.firebase.google.com
- Registered Android app with correct package name
- Downloaded and added `google-services.json` to `android/app/`
- Updated Gradle files:

`android/build.gradle:`

```
classpath 'com.google.gms:google-services:4.3.15'
```

- `android/app/build.gradle:`
 apply plugin: 'com.google.gms.google-services'

⚙️ Firebase Authentication Setup

- Enabled **Phone Authentication**
- Added test phone numbers for development
- Phone Auth Flow: Phone → OTP → Password setup → Save to Firebase

📌 Android Permissions

Added the following permissions in `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

```
<uses-permission  
android:name="android.permission.ACCESS_BACKGROUND_LOCATION"/>
```



Cursor AI Prompt-Driven Tasks(Prompt Strategy)

Use the following **prompts in Cursor AI** at each step. Run these **one at a time** and follow the manual instructions if needed.



Prompt 1: Clean Architecture with GetX

Create a clean architecture folder structure using GetX in Flutter 3.27.2. Use a feature-first approach. Include folders: `lib/core` (bindings, routes, utils), and `lib/features` (auth, attendance, calendar, more). Inside each feature folder, create `data`, `domain`, and `presentation` subfolders. Add placeholder files like controllers, views, and models. Use snake_case filenames and follow best practices for modularity.

Manual Tasks:

- Add dependencies in `pubspec`.
- Configure Firebase (Google services JSON / plist)
- Enable Phone Auth in Firebase Console



Prompt 2: Firebase Initialization

Set up Firebase in Flutter 3.27.2 using Dart 3.6.1. Include `firebase_core`, `firebase_auth`, and `cloud_firestore` packages. Initialize Firebase in `main.dart`. Add splash screen logic to check if the user is logged in and route accordingly using GetX. Don't include platform config (`google-services.json` or plist).

Manual Work:

- Add Firebase configuration to Android/iOS
- Set the correct SHA key in the Firebase console

Prompt 3: Bottom Navigation Setup

Create a main app shell with a BottomNavigationBar containing three tabs: Home, Calendar, and More. Use GetX for page switching. Implement this inside `lib/features/navigation/presentation`. Each tab should load its respective screen using indexed navigation. Add placeholder widgets for each screen for now.

Prompt 4: Phone Auth (Sign Up)

Create a Sign-Up screen using Firebase phone authentication. Step 1: User enters phone number → Send OTP. Step 2: Show OTP input → Verify. Step 3: Show password input → Save phone, UID, password, and current GPS coordinates (lat/lon) to Firestore under a `users` collection. Follow the GetX pattern and split into controller, UI, and service.

Prompt 5: Login With Phone + Password

Create a Login screen with phone number and password fields. On login, fetch the user by phone number from Firestore and match the password. If correct, log in the user and navigate to Home. Use the GetX controller and route to the bottom nav shell after success. Include error validation.

Prompt 6: Forgot Password With OTP

Create a Forgot Password screen using Firebase. Step 1: Enter phone number → Send OTP. Step 2: Show OTP field → Verify. Step 3: Show new password and confirm password fields → If match, update password in Firestore for that phone number. Follow GetX pattern and modular architecture.

Prompt 7: Home Screen + Check-In / Check-Out

Create a Home screen with the following UI: Centered logo, user name, current date and time, Check-In button (if not checked in), Check-Out button (if already checked in, colored red), check-in/check-out status, and both registered and current latitude/longitude. Use `geolocator` package to get current GPS. Save attendance data (timestamp + location) to Firestore under `attendance` collection. Use GetX and clean architecture pattern.

Prompt 8: Calendar Screen with Color Indicators

Create a Calendar screen showing attendance records using Firestore. Use green for days with check-ins, red for missed check-ins, and white for future dates. Use the `table_calendar` package. Fetch attendance data for the current user and month from Firestore and display using a GetX controller.

Prompt 9: More Screen Features

Create a More screen with three options: (1) My Attendance: navigates to a list of dates, check-in, and check-out times from Firestore. (2) Change Password: Old password, new password, confirm password, and update logic. Match the old password from Firestore, then update with the confirmed new one. (3) Logout: Show confirmation dialog and log the user out from Firebase.

Prompt 10: Route Management + Bindings

Set up GetX routing for all screens: splash, login, signup, forgot password, home, calendar, more, and sub-screens like change password and attendance list. Use GetPage routing in `app_routes.dart`, and register dependencies using bindings for each screen.

Prompt 11: Phone Number Formatting (India only)

Update all phone number input fields in the Sign-Up, Login, and Forgot Password screens to automatically prefix +91 to the entered number if not already present. Ensure the final phone number conforms to E.164 format (+91xxxxxxxxx). Add validation to prevent duplicate +91 or incorrect formats. Show an error message if the input is shorter than 10 digits or contains non-numeric characters.

Implement the formatting and validation logic inside the GetX controller or a utility method, and reflect the changes in the UI text fields accordingly.

Example:

User input: 9876543210

Final string used: +919876543210

 Fix Auto Login Issue on App Restart

In the SplashController, add logic to determine whether the user has truly authenticated or if it's just a stale currentUser session. Confirm that the user has recently logged in or has active session metadata. If not, force a logout using `await _auth.signOut();` and redirect to the login screen.

Prompt 12: Registration Error Handling

I'm building a multi-step registration flow using GetX and Firebase Phone Auth. Please define or fix a RegistrationController in registration_controller.dart that includes:

Controller Requirements:

RxInt currentStep — to track steps (0 = phone, 1 = OTP, 2 = password)

RxString verificationError — for OTP verification error messages

void sendOTP(String phoneNumber) — initiate Firebase phone auth using +91 if not present

void verifyOTP(String otp, String verificationId) — complete verification and move to next step

void completeRegistration(String password) — complete registration by linking phone number with password

Error Handling:

Set verificationError.value when FirebaseAuthException occurs

Display a proper message if the phone number is not in E.164 format

Routing:

Ensure the OTP screen navigation uses AppRoutes.VERIFY_OTP (define the route if it doesn't exist)

Add OTP Verification Route and Update Registration Logic

Prompt 13: Refactor the authentication flow

Refactor the authentication flow to use Firebase Phone Authentication with OTP instead of phone number and password. Follow these exact screen and behavior requirements:

1. Flutter version: 3.27.2 (Dart 3.6.1).
2. Add a splash screen as the initial screen.'

Login Screen:

- Show a phone number TextField and a "Send OTP" button.
- Below it, add a "New User?" button that navigates to the Sign In screen.

Sign In Screen:

- Show a phone number TextField and a “Send OTP” button.
- Once OTP is sent, show a 4-digit OTP field and change the button to “Verify OTP”.
- After OTP verification, display “User Registered” and redirect to Home Screen.
- Save the phone number and current location (latitude & longitude) in Firestore under the users collection.

Home Screen (Post Login):

- Center logo, user name, and current date/time.
- Show a “Check In” button that captures the current location.
- After check-in, hide Check In and show “Check Out” button (red color).
- Below buttons: show check-in time, attendance status (blank → Checked In → Checked Out), registered lat/lng, and current lat/lng.

Calendar Screen:

- Green for days the user has checked in, red for missed, white for future days.

More Screen:

- My Attendance:
 - Opens a list of records (Date, Check In Time, Check Out Time).
- Update Phone Number:
 - Fields: old phone number, new phone number, OTP field.
 - On submit: send OTP to the new number and update the phone number in Firestore.
- Logout:
 - Show confirmation dialog with Yes/No.
 - On Yes: log out the user and return to the login.

Additional Requirements:

- Remove any use of password fields or password authentication.
- Use FirebaseAuth.instance.verifyPhoneNumber for OTP.

- Handle verification callbacks properly: codeSent, verificationCompleted, verificationFailed, codeAutoRetrievalTimeout.
- Implement a reusable controller/service class to manage the OTP auth flow and Firebase interactions using GetX.
- Use clean architecture and Firebase best practices throughout.

Prompt 14: LOCATION HANDLING:

- Create a reusable method, ensureLocationAccess() that:
 - a. Checks location permission using permission_handler.
 - b. If permission is not granted, request it.
 - c. If permanently denied, it shows a Snackbar prompting the user to enable it from app settings.
 - d. Checks if location services (GPS) are enabled using geolocator.
 - e. If services are disabled, it prompts the user to turn on location by navigating to settings.
 - f. Returns true only if both permission is granted and services are enabled.
 - g. Wrap the logic in try-catch and show relevant errors via Get. snackbar.

2. INTERNET CHECK:

- Create a utility method ensureInternetAccess() that:
 - a. Uses connectivity_plus to check internet availability.
 - b. If offline, shows a Snackbar or Dialog with message: "You are offline. Please turn on the Internet."
 - c. Returns true only if connected via Wi-Fi or mobile data.
 - d. Use StreamSubscription to listen to connection changes and notify the user when disconnected.

3. INTEGRATION:

- In all views that involve online interaction (login, sign up, OTP verification, check-in/out, update phone), call both:


```
if (!await ensureInternetAccess() || !await ensureLocationAccess()) return;
```


- Only proceed with backend or location logic if both checks pass.

4. USER EXPERIENCE:

- Show appropriate progress indicators or blocking UI while checking.
- Make sure Snackbar or error messages are user-friendly.

Additional configurations

- Firebase security rules for Firestore:
- **Attendance Collection** (`/attendance/{userId}`): Allows authenticated users to read and write only their own attendance documents.
- **Records Subcollection** (`/attendance/{userId}/records/{recordId}`): Allows authenticated users to read and write only their own attendance records.
- **Users Collection (Optional: `/users/{userId}`)**: Allows authenticated users to read and write only their own user documents.

```
rules_version = '2';

service cloud.firestore {

  match /databases/{database}/documents {

    // Rule for attendance collection

    match /attendance/{userId} {

      // Allow user to read/write only their own attendance documents

      allow read, write: if request.auth != null && request.auth.uid == userId;
      // Match subcollection 'records'

      match /records/{recordId} {

        allow read, write: if request.auth != null && request.auth.uid == userId;

      }

    }

    // Optional: rules for other collections, e.g. users

    match /users/{userId} {

      allow read, write: if request.auth != null && request.auth.uid == userId;

    }

  }

}
```

```
}  
  
}
```

Testing Checklist

- Phone input works with or without +91
 - OTP sent correctly
 - Location permission and prompt work as expected
 - Internet check shows snackbar when offline
 - User data is saved to Firestore after login
 - Attendance marking
-

Final Summary

This document demonstrates a fully structured approach to building a **real-world Flutter attendance tracking app**, using **modern best practices** like GetX and Clean Architecture. The project is built with maintainability, modularity, and scalability in mind and effectively uses **Cursor AI for automation and productivity**.
