# **Solar Project Application Documentation (Updated)**

This document provides an in-depth, updated overview of the Flutter-based Solar Project Application, covering its architecture, UI components, data models, backend services, and a detailed explanation of its data flow and new features.

## **1. Introduction**

The Solar Project Application is a mobile application designed to streamline the management of solar energy projects. It offers robust features for daily progress tracking of module mounting systems, real-time cable reconciliation, and a secure, multi-step authentication process. The application prioritizes a clean and responsive UI/UX, coupled with reliable data persistence and real-time capabilities powered by Firebase.

## **2. Application Architecture**

The application maintains a modular and layered architecture, promoting clear separation of concerns, reusability, and ease of maintenance.

lib/  
├── main.dart  
├── constants/  
│ ├── app\_constants.dart  
│ ├── category\_colors.dart  
│ └── mms\_material\_quantities.dart (NEW)  
├── models/  
│ ├── cable\_entry.dart  
│ ├── icr\_info.dart  
│ └── mounting\_progress\_item.dart  
├── providers/  
│ └── todo\_provider.dart  
├── screens/  
│ ├── auth\_screen.dart  
│ ├── auth\_wrapper.dart  
│ ├── cable\_schedule\_screen.dart  
│ ├── home\_screen.dart  
│ ├── icr\_info\_screen.dart (NEW)  
│ ├── info\_screen.dart  
│ ├── module\_mounting\_screen.dart  
│ ├── module\_reconciliation\_screen.dart  
│ ├── sign\_in\_screen.dart (NEW)  
│ ├── sign\_up\_screen.dart (NEW)  
│ └── splash\_screen.dart  
├── services/  
│ ├── auth\_service.dart (NEW)  
│ ├── firestore\_service.dart  
│ └── scope\_calculation\_service.dart (NEW)  
├── utils/  
│ ├── date\_utils.dart  
│ └── dialog\_utils.dart  
├── widgets/  
│ ├── cable\_reconciliation\_card.dart  
│ ├── cable\_summary\_widget.dart  
│ ├── category\_dropdown.dart  
│ ├── date\_picker\_field.dart  
│ ├── feature\_card.dart  
│ ├── filter\_buttons.dart  
│ ├── progress\_input\_row.dart  
│ ├── responsive\_layout.dart  
│ └── todo\_list\_item.dart  
└── extensions/  
 └── color\_extension.dart

### **2.1 Core Directories and Their Responsibilities**

* **lib/main.dart**: The application's entry point. It handles global Firebase initialization, defines the application's theme, and sets the initial route to AuthWrapper to manage the authentication flow.
* **lib/constants/**: Stores static, unchanging data, now including MmsMaterialQuantities which defines the material quantities per table for scope calculation.
* **lib/models/**: Defines the data structures (Plain Old Dart Objects - PODOs) used throughout the application. Each model includes fromMap() and toMap() methods for JSON serialization/deserialization, crucial for Firebase Firestore integration. New timestamp fields are added here.
* **lib/providers/**: (Contains todo\_provider.dart from a previous feature. Its relevance to the current solar app features is minimal and can be removed if not repurposed). This directory typically holds state management logic.
* **lib/screens/**: Contains the main UI pages (screens) of the application. Each file represents a distinct view or user interaction flow, including new authentication and registration steps.
* **lib/services/**: Encapsulates business logic related to external services, primarily Firebase Authentication and Firestore database operations. This layer abstracts away backend complexities and now includes scope calculation logic.
* **lib/utils/**: Provides utility functions and helper classes (e.g., date formatting, dialogs).
* **lib/widgets/**: Houses reusable UI components used across multiple screens for consistency and code reusability.
* **lib/extensions/**: Contains Dart extensions to add new functionality to existing classes.

## **3. Application Components**

### **3.1 Screens**

* **SplashScreen**:
  + **Purpose**: The initial loading screen, providing a branded welcome.
  + **UI**: Features a light sky-blue background, a central card with "ReNew" branding (dark green 'Re', light green 'New'), "Daily Progress Report," and "Vendor" text. Uses GoogleFonts for a modern look.
  + **Functionality**: Displays for 2 seconds with an AnimatedOpacity fade-in effect, then navigates to AuthScreen.
* **AuthWrapper**:
  + **Purpose**: Acts as the primary routing decision point after the splash screen. It determines the next screen based on the user's authentication status and the completion of their ICRInfo.
  + **Functionality**:
    - Listens to FirebaseAuth.instance.authStateChanges() to track user login status.
    - If ConnectionState.waiting, it shows the SplashScreen.
    - If a user is authenticated, it performs a FutureBuilder check using FirestoreService().doesIcrInfoExist(userId).
    - If ICRInfo exists, it navigates to HomeScreen.
    - If ICRInfo is missing, it navigates to IcrInfoScreen, enforcing the completion of registration.
    - If no user is authenticated, it navigates to AuthScreen.
* **AuthScreen**:
  + **Purpose**: Presents users with options to sign in or sign up.
  + **UI**: Displays the "ReNew" branding, a welcome message, and two prominent buttons: "Sign In" and "Sign Up".
  + **Functionality**: Navigates to SignInScreen or SignUpScreen respectively.
* **SignInScreen**:
  + **Purpose**: Allows existing users to log in.
  + **UI**: A form with Email and Password input fields, a "Sign In" button, and a link to SignUpScreen. Includes a loading indicator.
  + **Functionality**: Authenticates users via AuthService.signInWithEmailAndPassword(). Upon success, the AuthWrapper (listening to auth state changes) handles the subsequent navigation to HomeScreen or IcrInfoScreen.
* **SignUpScreen**:
  + **Purpose**: Allows new users to create an account.
  + **UI**: A form with Email, Password, and Confirm Password input fields, a "Sign Up" button, and a link to SignInScreen. Includes a loading indicator.
  + **Functionality**: Creates a new user account via AuthService.signUpWithEmailAndPassword(). **Crucially, upon successful account creation, it immediately navigates (pushReplacement) to IcrInfoScreen to collect mandatory profile information, making it a two-step registration process.**
* **IcrInfoScreen (New)**:
  + **Purpose**: Collects mandatory ICR (Installation Completion Report) details. This is the second, crucial step of user registration.
  + **UI**:
    - **Theme**: Features a light greenery color palette (#E8F5E9 background) with subtle wave patterns (CustomPaint).
    - **Input Fields**: Location (dropdown 1-56), Contact (10-digit numeric), 1200 GC (Full Table, Half Table - numeric), 500 GC (Full Table, Half Table - numeric), Dummy (numeric).
    - **Styling**: Input fields have rounded borders (radius 12), subtle shadows, and floating labels. Focused borders highlight in a fresh green (#66BB6A).
    - **Layout**: Fields are logically grouped into containers for 1200 GC and 500 GC. A placeholder is included for a future "Add Your ICR Drawing Here" feature.
  + **Functionality**:
    - Uses GlobalKey<FormState> for comprehensive form validation, ensuring all required fields are filled and valid.
    - **Data Storage**: On "Complete Registration" (after validation), it constructs an IcrInfo object (setting createdAt: DateTime.now()) and saves it to Firestore via FirestoreService.saveIcrInfo().
    - **Scope Calculation**: It then uses ScopeCalculationService.calculateMmsScope() to compute the total material scope based on the entered full/half table counts.
    - **Scope Persistence**: The calculated scope is saved to Firestore via FirestoreService.saveMmsScope().
    - **MMS Item Initialization**: It calls FirestoreService.initializeMountingProgressItemsWithScope() to ensure all MMS items in the ModuleMountingScreen have their totalScope correctly set based on the calculated values.
    - **Navigation**: After successful data storage and scope calculation, it navigates (pushReplacement) to the HomeScreen.
* **HomeScreen**:
  + **Purpose**: The main dashboard, providing access to different project modules.
  + **UI**: Displays four distinct features as tappable FeatureCard widgets: "Module Mounting System", "Cable Schedule", "Module Reconciliation", and "Info".
  + **Functionality**: Navigates to the respective screens upon tapping a card.
* **ModuleMountingScreen**:
  + **Purpose**: Allows users to update the daily progress of Module Mounting System components.
  + **UI**: A form displaying a list of items (e.g., Rafter, Purlin-1, Purlin-4) with "Today's Progress" (input), "Cumulative Progress" (read-only), and "Total Scope" (read-only).
  + **Functionality**:
    - Fetches MountingProgressItem data from Firestore in real-time using StreamBuilder. The totalScope for each item is dynamically loaded from the values calculated during ICR registration.
    - Allows users to input "Today's Progress" for multiple items simultaneously.
    - **Fix for Multiple Inputs**: TextEditingControllers are carefully managed using putIfAbsent to ensure user input is not lost on StreamBuilder rebuilds.
    - On "Save Daily Progress", it iterates through all entered values, updates the cumulativeProgress for each item, sets lastUpdated: DateTime.now(), and saves the modified MountingProgressItems to Firestore via FirestoreService.saveMountingProgressItem().
* **CableScheduleScreen**:
  + **Purpose**: Facilitates real-time reconciliation of scheduled vs. actual cable cut lengths and minimizes wastage.
  + **UI**: Displays each cable entry in a CableReconciliationCard with details, an input for Actual Cut Length, and computed Wastage with color highlighting (Red: negative, Green: positive, Grey: zero). A CableSummaryWidget at the bottom shows overall totals.
  + **Functionality**: Fetches CableEntry data from Firestore in real-time. As the user types, individual wastage and overall summary update instantly. Changes are saved to Firestore via FirestoreService.saveCableEntry().
* **InfoScreen**:
  + **Purpose**: Provides information about the vendor and offers a logout option.
  + **UI**: Displays "Vendor Form Name" (using the user's email), "Working Block (Location)", "Mobile Number", "1200 GC Table Count" (Full/Half), "500 GC Table Count" (Full/Half), and "Dummy Value". All these are presented in well-styled info cards. A prominent "Logout" button is at the bottom.
  + **Functionality**:
    - Fetches ICRInfo from Firestore using a StreamBuilder to display vendor-specific details dynamically.
    - The "Logout" button calls AuthService.signOut() and then navigates (pushAndRemoveUntil) back to the AuthScreen, clearing the navigation history.
* **ModuleReconciliationScreen**: A placeholder screen for future module reconciliation features.
* **InfoScreen**: A placeholder screen for general application information.

### **3.2 Models**

* **MountingProgressItem (lib/models/mounting\_progress\_item.dart)**:
  + Properties: name (String), todayProgress (double), cumulativeProgress (double), totalScope (double), **lastUpdated (DateTime?)**.
  + Methods: fromMap(), toMap() for Firestore serialization; addTodayProgress() (updates cumulativeProgress and sets lastUpdated).
* **CableEntry (lib/models/cable\_entry.dart)**:
  + Properties: scbNo (String), icrNo (String), inverterNo (String), scheduledLength (double), actualCutLength (double).
  + Methods: fromMap(), toMap() for Firestore serialization; wastage (getter), getWastageColor() for UI logic.
* **IcrInfo (lib/models/icr\_info.dart)**:
  + Properties: location (int), contact (String), gc1200 (Map<String, int>), gc500 (Map<String, int>), dummy (int), **createdAt (DateTime?)**.
  + Methods: fromMap(), toMap() for Firestore serialization.

### **3.3 Services**

* **AuthService (lib/services/auth\_service.dart)**:
  + **Purpose**: Centralized handling of Firebase Authentication operations.
  + **Methods**: user (Stream for auth state), signInWithEmailAndPassword(), signUpWithEmailAndPassword(), signOut().
  + **Error Handling**: Provides specific error messages for common Firebase authentication exceptions.
* **FirestoreService (lib/services/firestore\_service.dart)**:
  + **Purpose**: Manages all interactions with Firebase Firestore, abstracting database operations.
  + **Methods**:
    - userId (getter): Retrieves the current authenticated user's UID.
    - getMountingProgressItems() (Stream): Fetches real-time mounting progress items.
    - saveMountingProgressItem(item) (Future): Saves/updates a mounting progress item, **setting lastUpdated timestamp**.
    - initializeDefaultMountingProgressItems() (Future): Initializes default MMS items with totalScope: 0.0 and lastUpdated timestamp if the collection is empty.
    - initializeMountingProgressItemsWithScope(calculatedScope) (Future): **Crucial for scope**. Updates or creates MMS items with their totalScope from the calculated scope and sets lastUpdated timestamp.
    - getCableEntries() (Stream): Fetches real-time cable entries.
    - saveCableEntry(entry) (Future): Saves/updates a cable entry.
    - initializeDefaultCableEntries(defaultEntries) (Future): Populates default cable entries.
    - saveIcrInfo(icrInfo) (Future): Saves IcrInfo for the user, **setting createdAt timestamp**.
    - doesIcrInfoExist(userId) (Future): Checks for the existence of IcrInfo for a user.
    - getIcrInfo() (Stream): **New**. Fetches real-time IcrInfo for the current user.
    - saveMmsScope(scopeData) (Future): **New**. Saves the calculated MMS scope map.
    - getMmsScope() (Stream): **New**. Fetches the calculated MMS scope map.
* **ScopeCalculationService (lib/services/scope\_calculation\_service.dart) (New)**:
  + **Purpose**: Encapsulates the business logic for calculating the total scope of MMS materials.
  + **Methods**:
    - calculateMmsScope(IcrInfo icrInfo): Takes IcrInfo (specifically, FullTable and HalfTable counts from both GC types) and MmsMaterialQuantities to compute the total required quantity for each MMS material. Returns a Map<String, double>.

### **3.4 Widgets (Reusable UI Components)**

* **FeatureCard**: Generic card for dashboard features.
* **ProgressInputRow**: Displays a single MMS item's progress.
* **CableReconciliationCard**: Displays a single cable entry for reconciliation.
* **CableSummaryWidget**: Displays overall cable reconciliation totals.
* FilterButtons, CategoryDropdown, DatePickerField, ResponsiveLayout, TodoListItem: (From previous features, their direct use in the current solar app features might be limited).

### **3.5 Utilities & Extensions**

* **DateUtil**: Helper methods for date operations.
* **DialogUtils**: Utility for custom AlertDialogs.
* **ColorExtension**: Extension for color manipulation.

## **4. Backend Details (Cloud)**

The application relies entirely on **Firebase** for its backend infrastructure, providing real-time data synchronization, robust authentication, and scalable storage.

### **4.1 Firebase Authentication**

* **Purpose**: Manages user accounts and secure authentication.
* **Method Used**: **Email/Password Authentication**. This allows users to create accounts and log in using standard email and password credentials.
* **User Identification**: Each authenticated user is assigned a unique User ID (UID) by Firebase, which is used to segregate and secure their data in Firestore.

### **4.2 Cloud Firestore**

* **Purpose**: A flexible, scalable NoSQL cloud database used to store all dynamic application data.
* **Data Structure and Access**: All user-specific data is organized under a top-level users collection, with each user having their own document identified by their Firebase User ID (UID).
  + **Base Path**: users/{userId} (where {userId} is the authenticated user's UID).
  + **ICR Information**:
    - **Path**: users/{userId}/icrInfo/user\_icr\_data
    - **Details**: Stores the IcrInfo object, which includes Location, Contact, 1200GC and 500GC table counts, and Dummy value.
    - **Timestamps**: Includes a createdAt field, set to DateTime.now() when the IcrInfo is first saved during registration.
    - **Access**: Written by FirestoreService.saveIcrInfo(). Checked for existence by FirestoreService.doesIcrInfoExist(). Read by FirestoreService.getIcrInfo() (stream).
  + **Module Mounting Progress**:
    - **Path**: users/{userId}/mountingProgress/{itemName}
    - **Details**: Stores MountingProgressItem objects (e.g., 'Rafter', 'Purlin-1'). Each item's name is used as its document ID.
    - **Timestamps**: Includes a lastUpdated field, set to DateTime.now() whenever the todayProgress is added and the item is saved.
    - **Scope Integration**: The totalScope field for each item is dynamically set based on calculations from IcrInfo.
    - **Access**: Fetched via FirestoreService.getMountingProgressItems() (real-time stream). Saved via FirestoreService.saveMountingProgressItem(). Initialized/updated with calculated scope via FirestoreService.initializeMountingProgressItemsWithScope().
  + **Calculated MMS Scope**:
    - **Path**: users/{userId}/mmsScope/calculated\_scope
    - **Details**: Stores a single document (calculated\_scope) containing a Map<String, double> that holds the total calculated scope for each MMS material (e.g., {'Rafter': 100.0, 'Purlin-1': 80.0}).
    - **Access**: Written by FirestoreService.saveMmsScope(). Read by FirestoreService.getMmsScope() (stream).
  + **Cable Entries**:
    - **Path**: users/{userId}/cableEntries/{scbNo}
    - **Details**: Stores CableEntry objects. Each scbNo is used as its document ID.
    - **Access**: Fetched via FirestoreService.getCableEntries() (real-time stream). Saved via FirestoreService.saveCableEntry().

### **4.3 Firestore Security Rules**

**Critical for ensuring data privacy and integrity.** The following rules are implemented to enforce that:

* Only authenticated users can read or write data.
* Users can *only* read from and write to documents within their *own* dedicated users/{userId}/ path. They cannot access or modify other users' data.

rules\_version = '2';  
service cloud.firestore {  
 match /databases/{database}/documents {  
 // Allow read/write for authenticated users only to their own user data  
 match /users/{userId}/{document=\*\*} {  
 allow read, write: if request.auth != null && request.auth.uid == userId;  
 }  
 }  
}

## **5. Key Data Flows**

### **5.1 User Registration and Profile Completion (Two-Step Process)**

1. **App Launch & Splash**: User opens the app, sees SplashScreen.
2. **Authentication Check (AuthWrapper)**:
   * AuthWrapper checks FirebaseAuth.instance.authStateChanges().
   * **If User Not Authenticated**: AuthWrapper directs to AuthScreen.
     + User selects "Sign Up" and navigates to SignUpScreen.
     + User enters Email and Password, then taps "Sign Up".
     + AuthService.signUpWithEmailAndPassword() creates the Firebase Auth user.
     + SignUpScreen immediately navigates (pushReplacement) to IcrInfoScreen.
     + User fills out ICRInfoScreen (Location, Contact, Table Counts, Dummy).
     + User taps "Complete Registration".
     + IcrInfoScreen validates inputs.
     + IcrInfoScreen creates an IcrInfo object, setting createdAt: DateTime.now().
     + FirestoreService.saveIcrInfo() stores this IcrInfo at users/{userId}/icrInfo/user\_icr\_data.
     + ScopeCalculationService.calculateMmsScope() computes the material scope based on table counts from IcrInfo.
     + FirestoreService.saveMmsScope() stores the calculated scope at users/{userId}/mmsScope/calculated\_scope.
     + FirestoreService.initializeMountingProgressItemsWithScope() populates/updates users/{userId}/mountingProgress collection, setting the totalScope for each MMS item based on the calculated scope and adding lastUpdated timestamps.
     + IcrInfoScreen navigates (pushReplacement) to HomeScreen.
   * **If User Authenticated (Subsequent Login)**:
     + AuthWrapper detects a logged-in user (userId).
     + It calls FirestoreService.doesIcrInfoExist(userId).
     + **If ICRInfo exists**: AuthWrapper navigates to HomeScreen.
     + **If ICRInfo is missing**: AuthWrapper navigates to IcrInfoScreen, forcing completion of the profile.

### **5.2 Module Mounting Daily Progress Tracking**

1. **Screen Load**: ModuleMountingScreen loads.
2. **Data Fetching**: A StreamBuilder listens to FirestoreService.getMountingProgressItems(), fetching real-time MountingProgressItem data for the current user. Each item includes its totalScope (calculated during ICR registration).
3. **User Input**: User enters values into "Today's Progress" fields for one or more items. TextEditingControllers are retained across rebuilds to preserve unsaved input.
4. **Save Progress**: User taps "Save Daily Progress".
5. **Validation & Update**: The form validates inputs. For each item with a valid, non-zero "Today's Progress":
   * The MountingProgressItem object's todayProgress is set.
   * item.addTodayProgress() is called, which updates cumulativeProgress and sets lastUpdated: DateTime.now().
   * FirestoreService.saveMountingProgressItem(item) is called to persist the updated item to Firestore.
6. **Real-time Reflection**: As items are saved, the Firestore stream emits new data, causing the StreamBuilder to rebuild and the "Cumulative Progress" and "Total Scope" (if updated by ICR info) to reflect the latest values.

## **6. Responsive and Adaptive UI**

The application is built with responsiveness as a core principle:

* **Global ThemeData**: Defined in main.dart, providing consistent styling across all screens and adapting to system themes.
* **LayoutBuilder**: Used in ModuleMountingScreen and ProgressInputRow to dynamically adjust column widths and element spacing based on available screen size.
* **SingleChildScrollView**: Ensures forms (IcrInfoScreen, SignInScreen, SignUpScreen) are scrollable on smaller screens or when the keyboard is active, preventing overflow.
* **Flexible Sizing**: Widgets utilize Expanded and Flexible within Row and Column to distribute space proportionally, ensuring elements scale gracefully.
* **Max Width Constraints**: Components like the splash screen card and grouped input containers use maxWidth constraints to maintain aesthetic proportions on larger displays while scaling down for smaller screens.

This documentation serves as a comprehensive guide to the current state and capabilities of your Solar Project Application, detailing its features, architecture, and interaction with the Firebase backend.